SIX-MONTH REPORT

OCTOBER 1, 1994 to MARCH 31,1995

INVESTIGATION OF MODULARLY CONFIGURED ATTACHED PROCESSORS WITH INTELLIGENT MEMORIES

1. STATUS OF PROJECT

Objective 1 (register-level design of MCAP): Block diagrams of all MCAP components have been completed and documented (see Attachment A) by Glenn Gibson and an undergraduate student. These diagrams include all of the important registers in the components. More detailed designs based on these block diagrams are currently being done in order to fulfill the objectives below.

Objective 2 (architecture/algorithm case studies): The simulator software package is now complete and this study is now progressing at a stepped up pace. While learning the MCAP system, a doctoral student under the direction of Gibson has designed an architecture for performing matrix operations and has written programs for executing matrix multiplication and matrix inversion. If implemented in 1 micron CMOS, the architecture should have a peak performance of 250 MFlops/s. Simulations of matrix multiplication can be carried out at slightly more than 240 MFlops/s, giving a processor efficiency of approximately 96%. The processor efficiency for matrix inversion is currently less then 40%, but the program is being rewritten and we are hoping to attain an efficiency of over 60%. Another program for performing Gaussian elimination will then be written. Upon completion of this work an article will be submitted to the *Journal on Computer Simulation*. An undergraduate student, also under the direction of Gibson, has designed an architecture for performing iterative solutions to two-dimensional partial differential equations. Although he has completed a program for solving LaPlace's equation, his results are premature. Programs for other first and second

OFFICE OF THE UNDER SECRETARY OF DEFENSE (ACQUISITION)
DEFENSE TECHNICAL INFORMATION CENTER
CAMERON STATION
ALEXANDRIA, VIRGINIA 22304-6145

IN REPLY REFER TO DTIC-OCC

SUBJECT: Distribution Statements on Technical Documents

TO: OFFICE OF NAVAL RESEARCH
CORPORATE PROGRAMS DIVISION
ONR 353
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

1995 1027 025

1. Reference: DoD Directive 5230.24, Distribution Statements on Technical Documents, 18 Mar 87.

2. The Defense Technical Information Center received the enclosed report (referenced below) which is not marked in accordance with the above reference.

SIX-MONTH REPORT
N00014-93-1-1343
TITLE: INVESTIGATION OF
MODULARLY CONFIGURED ATTACHED
PROCESSORS WITH INTELLIGENT
MEMORIES

3. We request the appropriate distribution statement be assigned and the report returned to DTIC within 5 working days.

4. Approved distribution statements are listed on the reverse of this letter. If you have any questions regarding these statements, call DTIC's Cataloging Branch, (703) 274-6837.

FOR THE ADMINISTRATOR:

1 Encl

GOPALAKRISHNAN NAIR
Chief, Cataloging Branch

FL-171
Jul 93

DISTRIBUTION STATEMENT A:

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

DISTRIBUTION STATEMENT B:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; (Indicate Reason and Date Below). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT C:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; (Indicate Reason and Date Below). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT D:

DISTRIBUTION AUTHORIZED TO DOD AND U.S. DOD CONTRACTORS ONLY; (Indicate Reason and Date Below). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT E:

DISTRIBUTION AUTHORIZED TO DOD COMPONENTS ONLY; (Indicate Reason and Date Below). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT F:

FURTHER DISSEMINATION ONLY AS DIRECTED BY (Indicate Controlling DoD Office and Date Below) or HIGHER DOD AUTHORITY.

DISTRIBUTION STATEMENT X:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DOD DIRECTIVE 5230.25, WITHHOLDING OF UNCLASSIFIED TECHNICAL DATA FROM PUBLIC DISCLOSURE, 6 Nov 1984 (Indicate date of determination). CONTROLLING DOD OFFICE IS (Indicate Controlling DoD Office).

The cited documents has been reviewed by competent authority and the following distribution statement is hereby authorized.

A

(Statement)

OFFICE OF NAVAL RESEARCH
CORPORATE PROGRAMS DIVISION
ONR 353
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

(Controlling DoD Office Name)

(Reason)

(Controlling DoD Office Address, City, State, Zip)

DEBRA T. HUGHES
DEPUTY DIRECTOR
CORPORATE PROGRAMS OFFICE

(Signature & Typed Name) (Assigning Office)

26 SEP 96

(Date Statement Assigned)

order equations will also be written. A doctoral student and two master's level students are, under the supervision of Sergio Cabrera, developing architectures and writing programs for performing FFTs and signal processing algorithms. This work began in January. Although it took the students some time to learn the MCAP design philosophy, their effort is progressing well now. Other journal submissions will be made as this work matures.

Objective 3 (two memory controller designs): Yu-Cheng Liu is directing a master's level student and an undergraduate student in this work. It began in earnest in November, after the block diagrams in Attachment A were finalized. The designs are at the logic level and are being done using the Mentor Graphics software package. The address generation and partition pattern logic for the single-access (S) and dual- access (D) components have been completed. Also, a preliminary study of an alternative design that uses shift registers has been done. Their work to date is being submitted to the '95 ISCA Int'l Conf. on Computer Appl. in Industry and Engr. in Honolulu.

Objective 4 (technology evaluations): This work involves two doctoral students and two master's level students and is being guided by Vijay Singh. Singh, Gibson, two doctoral students and one master's level student attended the multichip module conference, MCMC '95, in Santa Cruz, California, in January and presented a paper (see Attachment C). The presentation was made by one of the doctoral students, Buck Gremel. Also, Singh's group has produced a paper that has been accepted by the *Int'l Journal of Electronics* (see Attachment D) and has submitted a paper to the *IEEE J. on Components, Packaging and Manufacturing Tech.* To date, the group has concentrated its efforts on the MCM CMOS design of an architecture for performing matrix operations, but one of the master's level students has been investigating GaAs DCFL technology. Preliminary work on Wafer Scale Integration (WSI) has also been done by Yi-Chieh Chang and a graduate student under the supervision of Singh. They will study the WSI implementation of MCAPs this summer.

Objective 5 (simulator development): Although the simulator software package, SIMARC, is complete and currently in use, several enhancements are being made. These enhancements are

being done by a master's level student and an undergraduate student under the direction of Gibson. They primarily concern the display of simulation results and the ability to dynamically change an architecture's attributes while a simulation is being executed. It has been found that the principal difficulty in using the SIMARC package is in writing programs for the algorithms. The master's level student has also begun working on a graphically assisted assembler designed to alleviate this problem. The instruction set for MCAPs has been finalized and documentation of the SIMARC software began in December (see Attachment B) and will continue through next summer. A paper on this package will be presented at the '95 Simulation MultiConference in Phoenix in April (see Attachment E).

2. CURRENT LEVEL OF EFFORT

Although no theses were completed during this six-month period, the level of effort has increased dramatically. In addition to the paper presented in Santa Cruz, the four papers listed in the October 1, 1994, report were presented in Taiwan and San Francisco. At present, there are four doctoral stu-dents, eight master's level students and three undergraduates involved in the project. Of these, four of the graduate students are not currently supported (two are part-time students) and two undergraduates are being supported by stipends provided by the University. All others are supported by stipends and research assistantships through this grant.

3. PAPERS ACCEPTED FOR PUBLICATION (see attachments)

J. Singh, B.Gremel, V. Singh and G. Gibson, "Design Consid-erations for implementing a Modularly Configured Attached Processor in a Multi-Chip Module," *Proc. of MCMC '95*, Santa Clara, CA, Jan., 1995, pp. 62-65.

G. Gibson, A. Brito, Y. Chang, D. Saenz and E. Castro, "Simu-lation and Fast Prototyping of Modularly Configurable Attached Processors," *Proc. of 1995 Multiconference on High Performance Computing*, Phoenix, AZ, April, 1995.

J. Singh, B. Gremel, V. Singh, and G. Gibson, "Design Issues in a CMOS Implementation of a Modularly Configured Attached

Processor", accepted by *Int'l J. of Electronics.*

4. PAPERS SUBMITTED FOR PUBLICATION

J. Singh, S. Nagabathula, V. Singh and G. Gibson, "Comparative Evaluation of MCM and WSI Schemes for Implementing a Modularly Configured Attached Processor Architecture," submitted to *IEEE Trans. on Components, Packaging and Manufacturing Tech.--Part B: Adv. Packaging.*

Y. C. Liu, G. Gibson and S. Vaishampayan, "Intelligent Memory Controllers for Modularly Configured Attached Processors," submitted to *'95 Int'l Conf. on Computer Appl. in Industry and Engr.* in Honolulu, Hawaii.

ATTACHMENT A

CHAPTER 1
MCAP DEFINITION

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components. This set consists of two types of connections and ten types of components. The definitions of the connection and component types provide a standard set of rules that allow the components to be easily configured in different ways to construct attached processors that efficiently perform different sets of algorithms.

An MCAP is connected to its host using separate instruction and data streams. There is one instruction stream and it flows from the hosts memory to the MCAP's only instruction component. The instruction stream is depicted in Fig.1.1. It is assumed that, for each algorithm the MCAP is to execute, the instructions for the algorithm have been permanently stored as a subroutine in a ROM in the instruction component. The instructions sent from the host to the instruction component's RAM designate which algorithm is to be executed and the parameters needed by the algorithm(e.g., sizes and main memory locations of matrices). An algorithm is executed by drawing from the set of instructions received from both the host and subroutine. Using these instructions and the instruction component's register set and internal logic, an instruction stream, called the stream of external instructions, is produced and directed to the other component's in the MCAP. The stream of external instructions provides the other components with the information needed to perform the algorithm. Each external instructon is directed to a component by putting the component's identifying address (i.e., component number) on the address bus. The receiving component causes the contents of the instructon to be distributed to its appropriate registers. It is the contents of the component's registers that dictate its actions during the execution of an algorithm. For the given algorithm, the set of instructions that a component receives must be sufficient to fill the registers needed by the algorithm.

There may be multiple data streams between the host and MCAP and they are connections to one or more of the MCAP's memory subsystem components. The primary purpose of these components is to buffer data between the host and the MCAP, although they may also rearrange the data being input from or output to the host. Once the data has been put into the memory subsystem components, other MCAP components may be used to process the data and return

results to the memory system components, from which the results may be output to the host's main memory. All movement and processing of data to and from the host and within the MCAP is determined by the external instructions sent from the MCAP's instruction component to other components in the MCAP. These instructions set up memory-to-memory pipelines that route and process the data according to the algorithm to be performed.

All MCAP connections are unidirectional and asynchronous. There are two types of connections within an MCAP, instruction connections and data connections. A data connection consists of a data bus and a Request/Acknowledge (Req/Ack) pair. A transfer begins when the transmitting component puts the data on the data bus and activates the Req line. When the receiving component recieves the Req signal and accepts the data, it pulses the Ack line. The transfer is complete and the data and Req signals are dropped when the Ack pulse is received by the transmitting component. An instruction connection is similar except that it may be connected in more than one receiving component and includes an address bus. An address is put onto this bus at the same time an instruction is put onto the data bus. Only the addressed component can receive the data and return the Ack pulse.

The ten types of components fall into four categories and are summarized as follows:

Instruction (I)

Processor:

- Elementary–one input, one output(E)
- Two-input–two inputs, one output (T)
- Comparator–two inputs, one output plus special outputs (C)

Router:

- Join–multiple inputs, one output (J)
- Fork–one input, multiple outputs (F)
- Link–multiple inputs, multiple outputs (L)

Memory subsystem:

- RAM–one input, one output, no partitions (R)
- Single-access–one input, one output, has partitions (S)
- Dual-access–two inputs, two outputs, has partitions (D)

The letter used to indicate each type of component is given in parentheses. These component types are described in the subsections below.

1.1 Instruction Components

An MCAP contains exactly one instruction (I) component. A block diagram of an I component is given in Fig. 1.2 . As explained above, an I component receives instructions from the host and produces a stream of external instructions that are sent to the other components in the MCAP. The instructions sent from the host are put into the I components RAM. These instructions are then brought in through the input instruction queue and decoded. They include the parameter values needed by the algorithm to be executed and cause these values to be put into the register set. The last instruction received from the host causes the subroutine that executes the algorithm to be initiated. The instructions in the subroutine are then brought in through the input instruction queue. There are two types of instructions, internal instructions and external instructions. The internal instructions, the parameter values in the register set and the internal logic of the I component are used to produce other parameter values that are also stored in the register set. External instructions are those that are, perhaps, modified by the parameters in the register set and sent out to the other MCAP components via the output instruction queue and instruction connection. Internal instructions are capable of performing integer arithmetic and logical operations, subroutine calls and returns, looping, unconditional branches, and conditional branches based on flag signals received from the processing components. One of the processing components must be a comparator. As discussed below, a comparator is capable of determining a maximum or minimum of a sequence of numbers as well as comparing two values. Therefore, not only are flag signals sent from the comparator that indicate $>, <, \geq, \leq, =$ and $\neq$, but the comparator may also return the index within a sequence of a maximum or minimum values. Such indices may be used to modify external instructions, particularly the instructions for computing addresses.

In addition, the I component receives flags from the other processing components that indicate exceptional conditions (e.g., division by zero, overflows, underflows and so on). These flags are used to interrupt the host via the control bus when the MCAP aborts an algorithm.

1.2 Processor Components

The processor components are for performing unary and binary arithemtic/logic operations. There are three types of processor components. There are one-input elementary (E) components, two-input (T) components , and comparator (C) components. Figure 1.3 gives a block diagram, programmable register summary, and mode register definition for an E component. As seen from the block diagram an E component has an input instruction connection, an input data connection, an output data connection, and a set of flag lines. It consists of an input queue at its input connection, process logic for performing the required operation, and control logic that contains four programmable registers. The mode register dictates the actions taken by the component and it, along with the other three registers, determine the order and manner in which the data inputs are used and when the component must input more instructions.

A list of the currently available modes for an E component are given in Table 1.1. Eight bits in an E component's Mode register are used. Bit 0 indicates whether are not the component is being used to perform an accumulation (e.g., sum a column of numbers). This bit is needed only if the E component is part of a pipeline that can perform accumulations. When this bit is 1 the number in the NumOpsOut register is automatically modified according to the number of stages in the pipeline, thereby taking into account the final accumulation steps. Bits 1 through 4 indicate the components current function with bit 4 specifying whether an unary or a binary operation is to be conducted. If an E component is used for a binary operation, the first operand is transferred to a latch and the operation proceeds when the second operand arrives. It is possible for an E component to output a constant or perform a binary operation with one of the operands being held constant. Whether a constant is being used is determined by bit 5. If a constant is being is used it is possible to fill the latch by using an immediate instruction or by using the first datum that arrives at the input data queue. Bit 6 is used to indicate one of these two choices. Bit 7, 8 and 9 are not used. Bit 10 allows an E component to be put into primitive mode. In primitive mode, an E component does not accept additional instructions, but is simply a passive component that inputs and processes data as it becomes available.

Except for an E component in primitive mode or an I component, from the standpoint of any component an algorithm is broken into tasks and, for each task, the component must receive

a sequence of instructions that determines what it is to do to execute the task. When the component has completed a task, it must input another sequence of instructions. Instructions are first used to fill the Mode, Number of Repetitions (NumRepetitions) and Decrement Amount (DecAmt) registers. (If one or more of these registers is not filled, it's current contents are used). Then, when an instruction fills the Number of Operands Out (NumOpsOut) register, the component begins executing the task. The Number of Operands Out Constant (NumOpsOutConst) register is automatically loaded from the NumOpsOut register. As the task executes, it draws inputs from it's input data queue(s), processes them, and outputs any results. Each time there is an output, the NumOpsOut register is decremented. Each time the NumOpsOut register becomes 0 the NumOpsOutConst register is decremented by DecAmt and used to reload the NumOpsOut register. Also, the NumRepetitions register is decremented by 1. When both NumOpsOut and NumRepetitions become 0 the task is complete and the component must input another input sequence of instructions.

Figure 1.4 gives the block diagram, programmable register summary, and mode register definition for a T component. It is similar to an E component, but the T component has two input data connections and corresponding queues. A list of the currently available modes for a T component are given in Table 1.2. Ten bits in a T component's Mode register are used, with the first seven serving the same purposes as in an E component. However, if a constant is input through one of the input data conections, the connection it arrives on is indicated by bit 7. Bit 8 specifies whether one or both inputs are to be used, bit 9 specifies which input any nonconstant input value will arrive on.

Figure 1.5 gives the block diagram, programmable register summary and mode register definition for a C component. A C component is a T component that has two special sets of lines connecting it to the I component. There can be only one C component in an MCAP. As usual, its current function is determined by its mode. A list of the currently available modes for a C component are given in Table 1.3. One of its functions is to simply compare two inputs and set relational flags that are then transmitted to the I component over one set of the special lines. When performing comparisions there are no outputs other than the flag outputs. The C component can, however, also determine the maximum or minimum of a sequence of numbers.

In this case, the second set of special lines is used to transmit the position, or index, of the maximum or minimum within the sequence to the I component. If the maximum or minimum occurs more than once in the sequence, the index always points to the first occurence. If a maximum or minimum is being determined, then NumOpsOut is used to specify the length of the input sequence (i.e., NumOpsOut really indicates the number of operands input). Also, the maximum or minimum is output on the output data connection at the same time its index is output on the index lines.

1.3 Routing Components

Routing components are for directing data along the proper paths. There are three types of routing components, join (J) components with more than one input and one output, fork (F) components with one input and more than one output, and link (L) components with more than one input and more than one output.

Figure 1.6 shows the block diagram, programmable register summary, and mode register definition for a J component. It has an input instruction connection, multiple input data connections and an output data connection. It consists of a queue for each input, a bus, control logic containing the same individual programmable registers as a processing component and a programmable set of input pattern registers (InPattern). The registers in a pattern are summarized in Fig.1.7. The NumRepetitions, DecAmt, NumOpsOut and NumOpsOutConst registers are used as they are in the processor components. The sole purpose of the Mode register is to distinguish between no accumulation and accumulation. As with processor components, J and F components may be part of a pipeline capable of performing accumulation. A processor component pipeline that performs accumulations always has a J component at one of its input data connections, an F component at its output data connection, and a feedback data connection from the F component to the J component. These F and J components must increase NumOpsOut by an amount that depends on the number of stages in the pipeline. Also, the F component, which normally outputs to the feedback connection must output the final result to a different output connection. This allows the pipeline to accumulate partial results and then produce a final result and send the final result to its destination. In the no accumulation

mode, InPattern specifies the order in which the input connections are to be selected during the execution of the algorithm being programmed. This pattern is continually cycled through until NumOpsOut becomes 0. It may be reused by the next task or changed by a new instruction that resets InPattern.

A pattern includes two subcycles. The Pattern Array (PArray) set of registers shown in Fig.1.7 are set to a sequence of numbers that indicate input connections. The register Number1 indicates the number of connections selected during the first subcycle and the register Count1 indicates the number of registers from PArray that are to be included in the first subcycle. Number2 and Count2 serve the same purpose for the second subcycle. When the first subcycle is exhausted, the second subcycle is begun, and when the second subcycle is exhausted, a return is made to the first subcycle and so on.

For example, if Number1=4, Count1=3, Number2=5, Count2=2 and PArray contains 2, 6, 4, 7 and 1, then the input connections are used in the order

2 6 4 2 7 1 7 1 7 2 6 4 2 7 1 7 1 7 ....

The sequence continues until NumOpsOut becomes 0.

Figures 1.8 and 1.9 correspond to the F and L components, respectively. An F component, because it has only one input and multiple outputs, has a set of output pattern registers, OutPattern, instead of an input set, and an L component has both an InPattern and OutPattern. Both InPattern and OutPattern are continually cycled through and determine the order in which the input or output connections are used. An L component cannot be part of a pipeline and does not need a mode register.

In addition both F and L components contain a set of broadcast registers denoted BcPattern. If an entry in an output pattern is all ones, then the entry will not be used as an output connection number, but there will be a *simultaneous* output to all output connections listed in BcPattern.

1.4 Memory Subsystem

There are three types of memory subsystem comoponents, RAM (R) components, single access (S) components, and dual-access (D) components. All memory subsystem components are for automatically retrieving operands from and storing results in their associated memory modules. All memory subsystem components have an output data connection and an input data connection. Therefore, they must be capable of handling both an output data stream and an input data stream. In addition, a D component includes a second pair of input and output connections. All memory subsystem components have a queue in each of their input and output data streams.

A significant difference between the memory subsystem components and the other components is that a Number of Operands In (NumOpsIn) register as well as a NumOpsOut register must be included. The NumOpsIn register serves the same purpose for the input data stream as NumOpsOut does for the output stream. Both NumOpsIn and NumOpsOut must be zero before new instructions can be distributed to the component's programmable registers.

Figure 1.10 gives the block diagram, programmable register summary, and mode register definition for the R component. An R component is primarily used for temporary storage or as a large queue. An R component has six modes related to the input and output of data. They are:

Input:	Data is only stored in memory (i.e., only the input data stream is used).
Output:	Data is only retrieved from memory (i.e., only the output data stream is used).
Input/Output:	Data is first input to memory and then output from memory. The input must stay ahead of the output.
Output/Input:	Data is first output from memory and then input to memory. The output must stay ahead of the input.
Input and Output:	Data is input and output with no regard as to which is done first.
Zero:	Put zeros in all memory locations.

Figure 1.11 gives a block diagram of an S component. An S component differs from an R

component in that it may be connected to more than one memory module and the memory as a whole may be divided into partitions, called S partitions, that consist of blocks of memory having consecutive addresses. The memory modules may be banked and/or interleaved (i.e., the high-order address bits specify the bank and the low-order bits specify the module within the bank). The partitions, because they occupy consecutive addresses, are spread across the modules and may even encompass more than one bank.

Because some S partitions are used for outputting from memory(i.e., providing the output data stream) and some are for inputting to memory (i.e., terminating the input data stream), there is a set of programmable registers referred to as the Output Partition Pattern, OutPartPat, that determines the pattern in which the partitions providing the output stream are accessed. Likewise, the Input Partition Pattern, InPartPat, determines the order in which the partitions providing the input stream are accessed. In both cases the patterns determine the partition sequence in the same way the connection sequences are determined by the routing components (see Fig.1.7).

Each S partition is accessed as a circular memory (i.e., the first location in the partition is considered to follow the last location in the partiton). A summary of the registers that define a partition and are used to determine the order in which the locations within a partition are accessed is given in Fig.1.12. A partition is defined by its Base register, that gives its base address, and Size register, that designates its size. From Fig 1.12(b) it is seen that each partition has a mode and can be put into any one of the first four modes permitted an R component. If the mode of a partition allows output, a window must be defined within the partition. All outputs must be from within the window and if there are inputs, they must be to locations outside the window. An exception occurs when a partition is in its input before output mode. In this case the window must be filled before output begins, but thereafter the input must be to outside the window. The initial base address of the window is the same as that of the partition. The window's base is incremented with each repetition of a window pattern, which is described below. The size of the window is defined by the contents of the WinSize register.

Within an S partition the sequence of input addresses is generated by

$I=0$

```
While NumOpsIn > 0 {
    Input address = B + I mod S
    Increment I by 1
    }
```

where B and S are the base and size of the partition.

The addresses used for outputting are generated according to the output pattern defined by the PatInc, NumReps1, RepInc1, NumReps2, and RepInc2 registers and the offset pattern. Let P, N_1, R_1, N_2 and R_2 be the contents of these registers respectively, and OS be the PArray in the offset pattern (see Fig. 1.7.). Also, let N_0 equal the initial NumOpsOut and N_3 equal the sum of Number1 and Number2 from the offset pattern. Assuming $N_1 > 0$ and $N_2 > 0$ then the sequence of output addresses is generated by

```
K0=K1 = K2 = M1 = M2 = M3 = 0
While M0 < N0{
    K1 = 0
    M1 = 0
    While M0 < N0 and M1 < N1{
        K2 = 0
        M2 = 0
        While M0 < N0 and M1 < N1 and M2 < N2{
            M3 = 0
            While M0 < N0 and M1 < N1 and M2 < N2 and M3 < N3{
                Select I from offset pattern
                Output address = B + ( K0 + K1 + K2 + OS[I] ) mod S
                Increment M0, M1, M2 and M3
                }
            IncrementK2 by R2
            }
        IncrementK1 by R1
        }
    IncrementK0 by P
    }
```

For example, suppose that the offset pattern in

Number1=4 Count1=3 Number2=Count2=0 PArray= {0,3,1}

and

B=0 S=12 N_0 =25 P=1 N_1=20 R_1=2 N_2=7 R_2=4.

Then N_3=4 and the sequence of addresses generated would be

0 3 1 0 4 7 5 2 5 3 2 6 9 7 4 7 5 4 8 11 1 4 2 1 5

External instructions must, of course, be sent to an S component to specify the mode, define the S partitions and specify the input and output partition patterns. The modes for the partitions are determined by the mode of the S component. For each partition that produces output, there must be external instructions for specifying the window size and overall output pattern.

The format of an S component mode instruction is given in Fig. 1.13. Bits 0 through 29 are divided into pairs with each pair specifying the mode of an S partition. Bits 0 and 1 specify the mode for partition 0 and so on. When a mode instruction is recognized, its lower 30 bits are seperated into pairs and the pairs are sent to the corresponding S partition mode registers. Bit 30 is put in Bit 0 of the S component's mode register. The format of this register is given in Fig. 1.14. Bit 30 is used to indicate whether only one data stream is to be used or both the input and output streams are to be used.

A block diagram of a D component is given in Fig. 1.15. In a D component there are two input streams and two output streams. The input stream logic in a D component is the same as in a S component, but it is replicated. Also, the output stream logic is the same, but replicated. The S partition logic is the same as in an S component.

The formats of the two D component mode instructions are shown in Fig. 1.16. The one shown in Fig. 1.16(a) provides pairs of bits for specifying the S partition modes for partitions 0 through 12. These pairs are in Bits 0 through 25. Bit 26 indicates that the attached processor output (AP out) stream is to be used when it is 1 and not to be used when it is 0. Similarly Bits 27, 28 and 29 indicate the use of the attached processor input (AP in) stream, host output (Host out) stream and host input (Host in) stream. The instruction in Fig. 1.16(b) gives the S partition mode pairs for partitions 13 through 27. Bits 26, 27, 28, and 29 are put in the four low-order bits of the D component's mode register. As with an S component, a mode instruction distributes the mode bits to the corresponding S partition mode registers. The format of this register is given in Fig.1.17.

1.5 Example Architechture

An MCAP for performing matrix operations is given in Fig. 1.18. The letter in each component gives the component's type. The MCAP includes a comparator (a C component) a negator/reciprocator (an E component), three four-stage adder/subtractors (a T component followed by three E components), three four-stage multipliers (a T component followed by three E components), several J, F and L components for routing the data and three memory subsystem components. The small rectangles inside the memory subsystem components represent data streams, address generators, memory buses and memory modules. The S component at the top is for temporarily storing data. The D component acts as a buffer, but is also used to rearrange and temporarily store data. The S component at the bottom is the MCAP's controller of the host's main memory and determines how the MCAP accesses main memory. The I component, instruction connections and host's connection to main memory are not shown.

Table 1.1: Summary of E Component Modes

Mode	Description
0xxx110fff0	Immediate constant is output NumOpsOut times.
0xxx010fff0	Constant is input and then output NumOpsOut times.
0xxx000fff0	Unary operations are performed on NumOps Out inputs and NumOpsOut results are output.
0xxx111fff0	Binary operations are performed using an immediate constant with NumOpsOut inputs and NumOpsOut results are output.
0xxx011fff0	Constant is input and then binary operations are performed using this input with NumOpsOut additional inputs to produce NumOpsOut outputs.
0xxx001fff0	Binary operations are performed on NumOpsOut pairs of successive inputs and NumOpsOut results are output.
1xxx000fffa	Component is put into primitive mode. For each input, a unary operation is performed and a result is output.

x – not used
f – function code bit assigned by designer.
a – if component is part of an accumulation pipeline it is 1: otherwise, it is 0.

Table 1.2: Summary of T Component Modes

Mode	Description
010110fff0	Immediate constant is cutput NumOpsOut times.
00c010fff0	Constant is input and then output NumOpsOut times.
v00000fff0	Unary operations are performed on NumOpsOut inputs and NumOpsOut results are output.
v00111fff0	Binary operations are performed using an immediate constant with NumOpsOut inputs and NumOpsOut results are output.
v0c011fff0	Constant is input and then binary operations are performed using this input with NumOpsOut additional inputs to produce NumOpsOut outputs. Only one input connection is used and v and c are equal.
v1c011fff0	Same as previous mode except that two inputs connections are used, one for the constant and one for the variables, and v and c are complements.
v00001fff0	Binary operations are performed on NumOpsOut pairs of successive inputs arriving on a single input connection and NumOpsOut results are output.
010001fff0	Binary operations are performed on NumOpsOut pairs of successive inputs and NumOpsOut results are output. Both input connections are used and for each operation one operand must arrive on one connection and the other operand must arrive on the other connection.
010001fff1	Component is to be used as a part of an accumulaltion pipeline. Both inputs are used as required by the pipeline.

f - function code bit assigned by designer.
v - number of the input connection to be used to input the variables.
c - number of the input connection to be used to input the constant.

Table 1.3: Summary of C Component Modes Description

Mode	Description
v0011001f0	Immediate constant is compared with NumOpsOut inputs. The minimum is output and the index of the first minimum is sent to the I component. If the immediate constant is equal to the minimum, the index is all 0's and the flags indicate that the immediate constant is equal to the minimum.
v0011010f0	Same as the first entry except that the maximum is found instead of the minimum.
v0011011f0	Same as the first entry except that both the maximum and the minimum are found and both extrema and their indices are output. The minimum and its index are output first.
v0000000f0	A pair of inputs are compared and only the flags are output. Only one input connection is used.
v0000001f0	The minimum of NumOpsOut inputs is found. The first minimum is output and its index is sent to the I component. Only one input connection is used.
v0000010f0	Same as the fifth entry except that the maximum is found instead of the minimum.
v0000011f0	Same as the fifth entry except that both the minimum and maximum are found and both extrema and their indices are output. The minimum and its index are output first.
01000000f0	The inputs on the two input connections are compared and only the flags are output.
v1c01001f0	Same as the first entry except that the constant arrives on the input connection not used by the variable input stream. v and c are complements.
v1c01010f0	Same as the second entry except that the constant arrives on the input connection not used by the variable input stream. v and c are complements.
v1c01011f0	Same as the third entry except that the constant arrives on the input connection not used by the variable input stream. v and c are complements.
01000100f0	NumOpsOut inputs arrive on each of the two input connections and the absolute values of the differences of the successive pairs of inputs are determined. These absolute values are compared with an immediate operand and the flags indicate whether are not all of them are less than or equal to the immediate operand. Only the flags are output.
v0000100f0	Same as the preceeding entry except that the NumOpsOut input pairs arrive on a single connection.

f - function code bit assigned by designer.
v - number of the input connection to be used to input the variables.
c - number of the input connection to be used to input the constant.

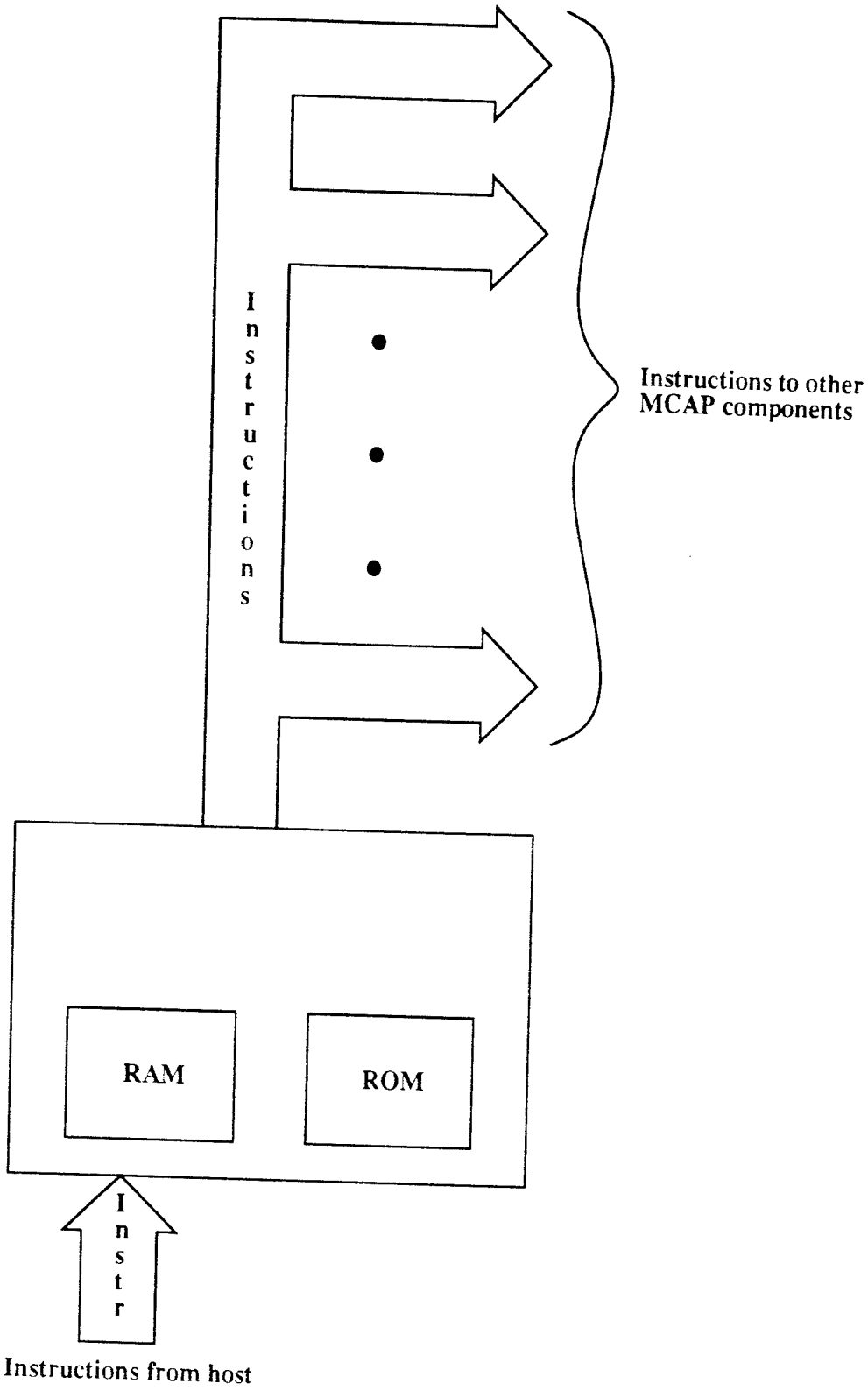


Fig. 1.1. Instruction stream.

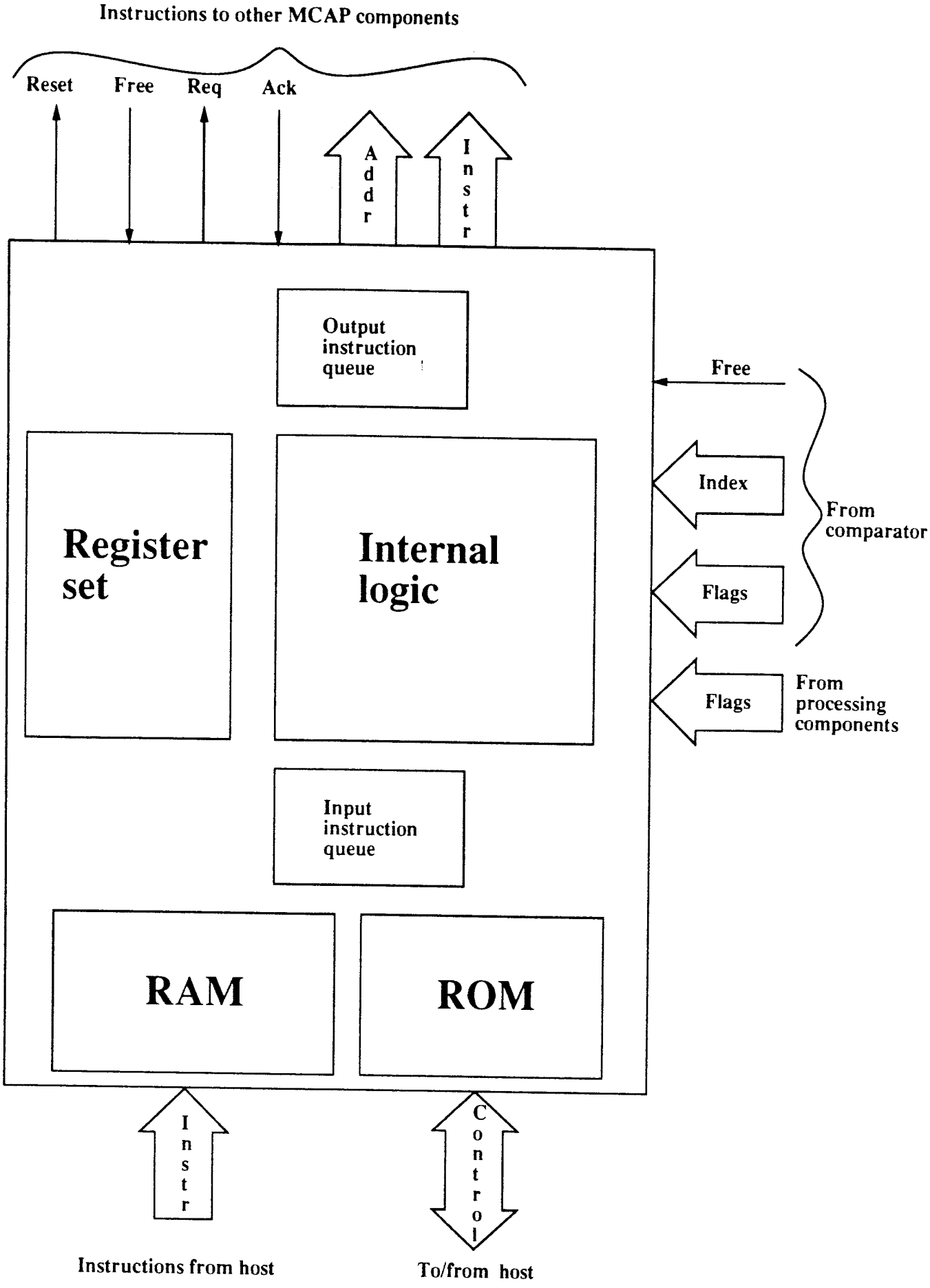
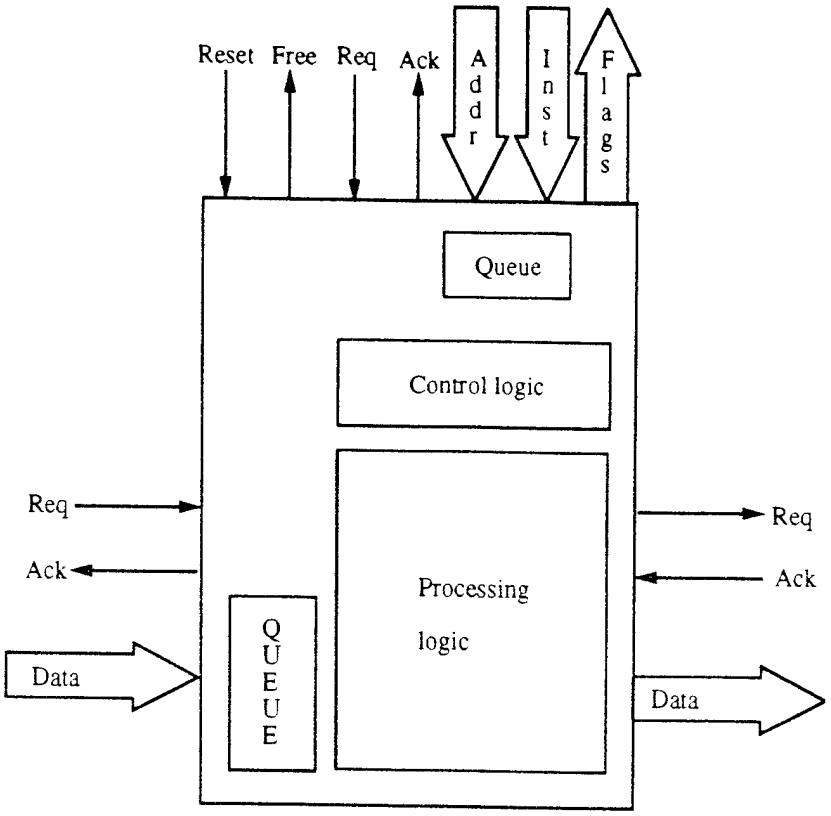
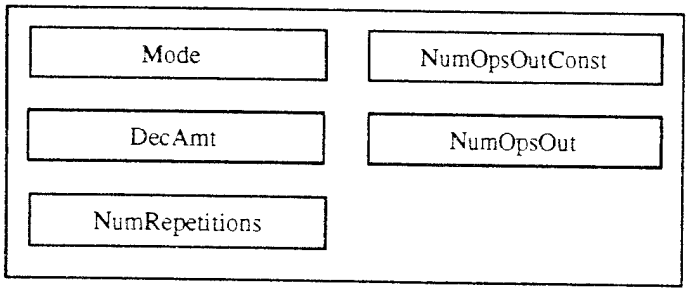


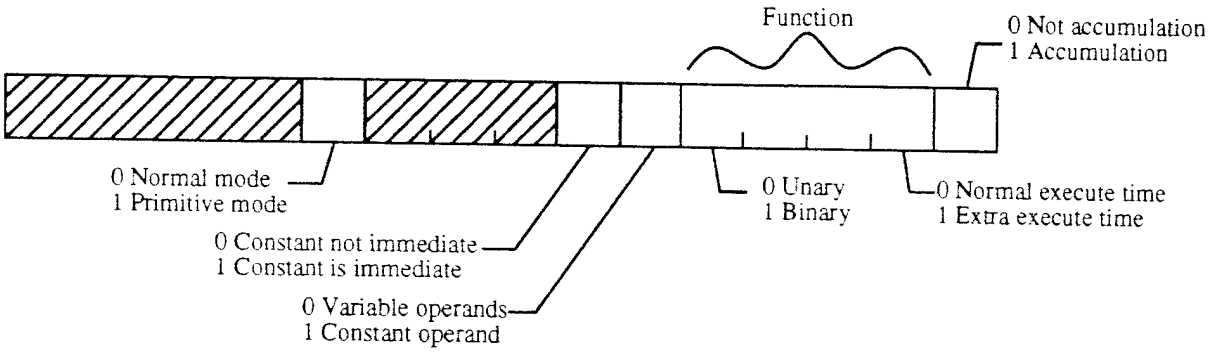
Fig.1.2. Instruction component.



(a) Block Diagram.

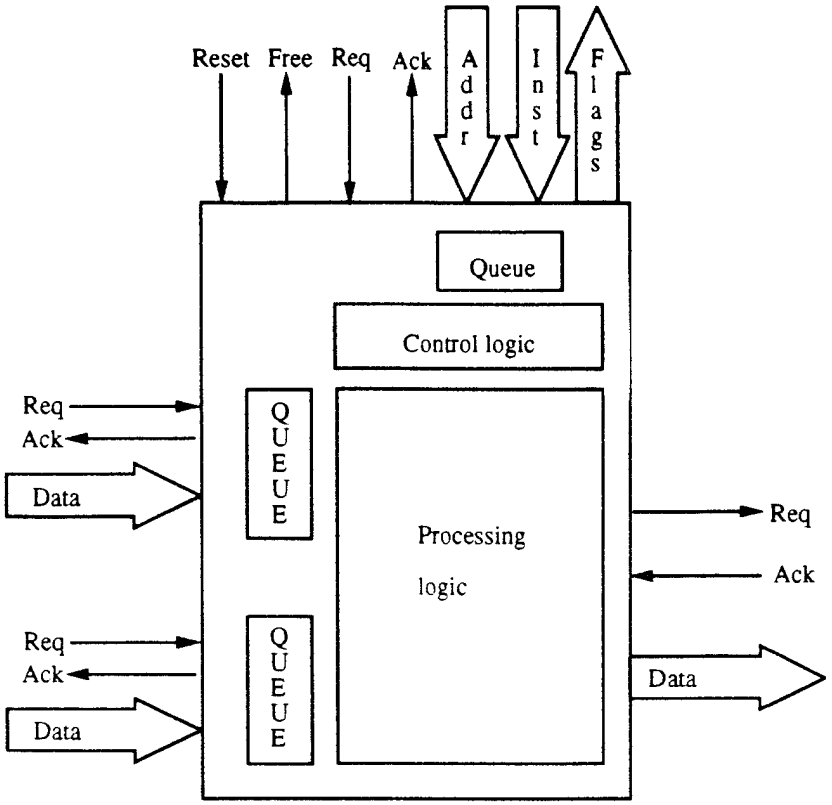


(b) Programmable registers.

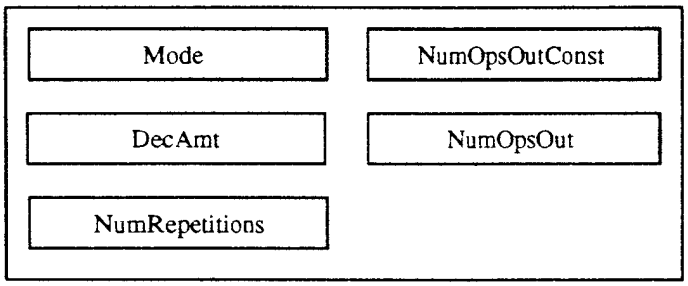


(c) Mode register.

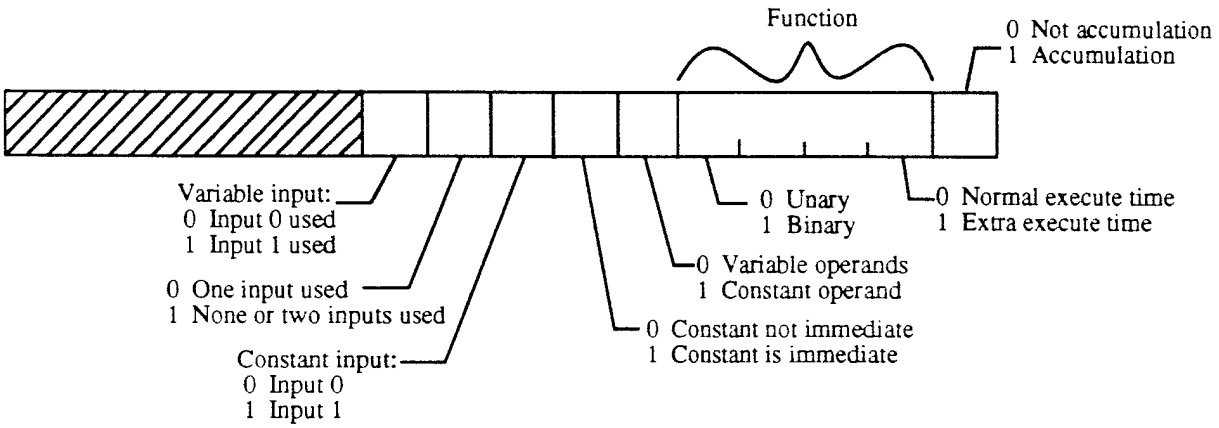
Fig. 1.3. E component



(a) Block Diagram.

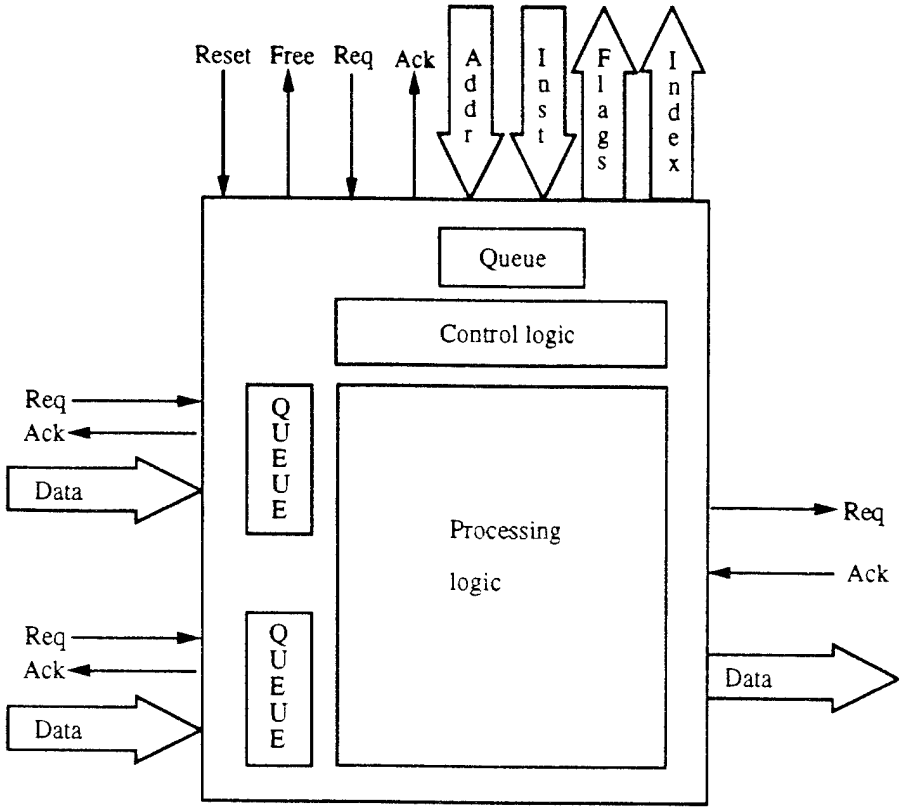


(b) Programmable registers.

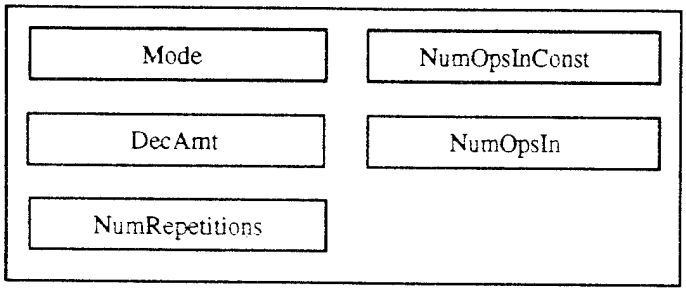


(c) Mode register.

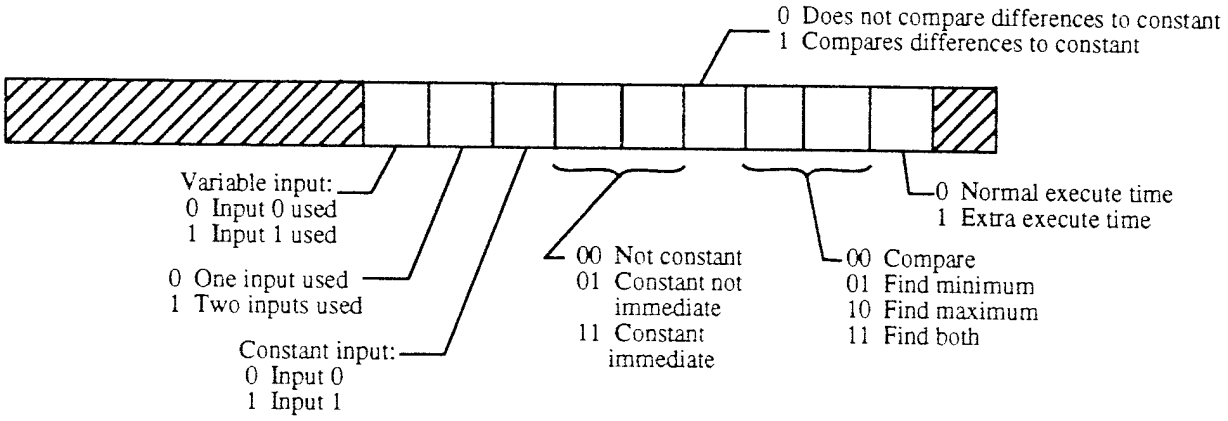
Fig. 4. T component



(a) Block Diagram.

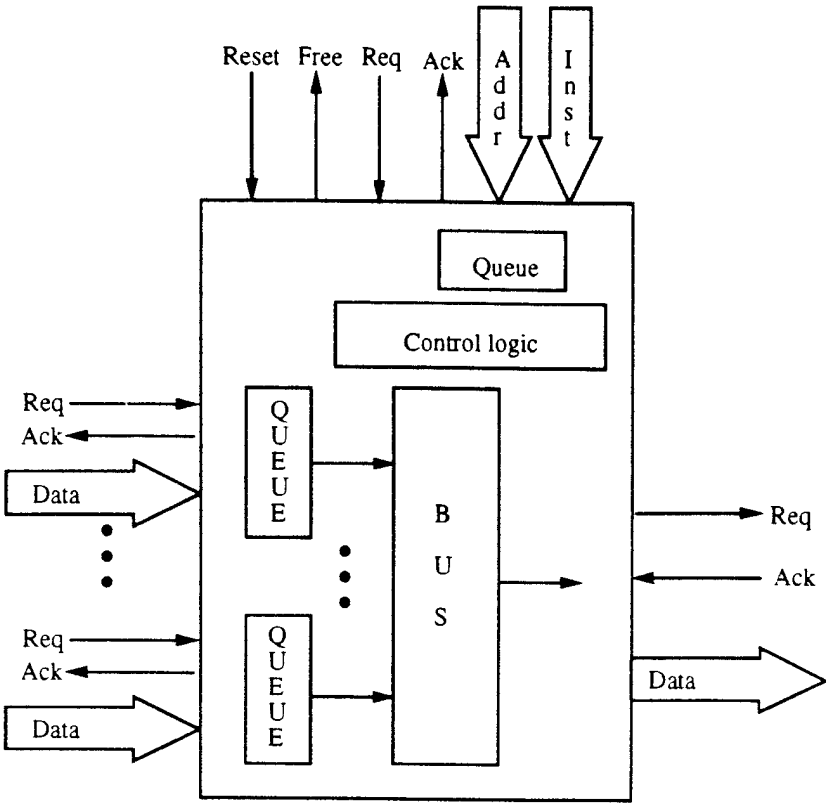


(b) Programmable registers.

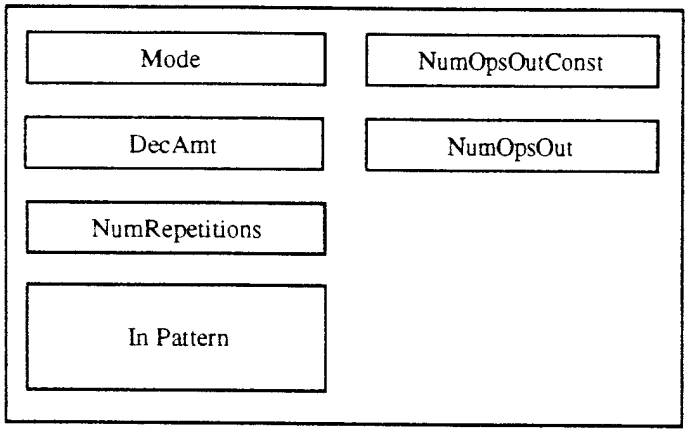


(c) Mode register.

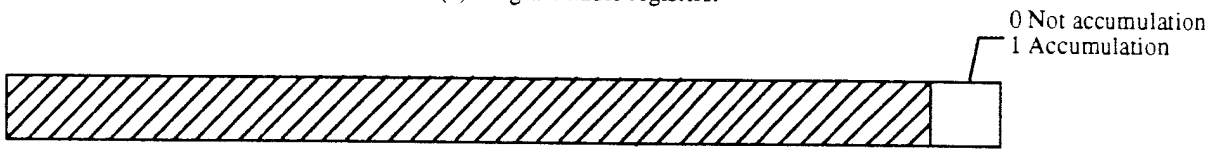
Fig.5. C component



(a) Block Diagram.



(b) Programmable registers.



(c) Mode register.

Fig. 1.6. J component

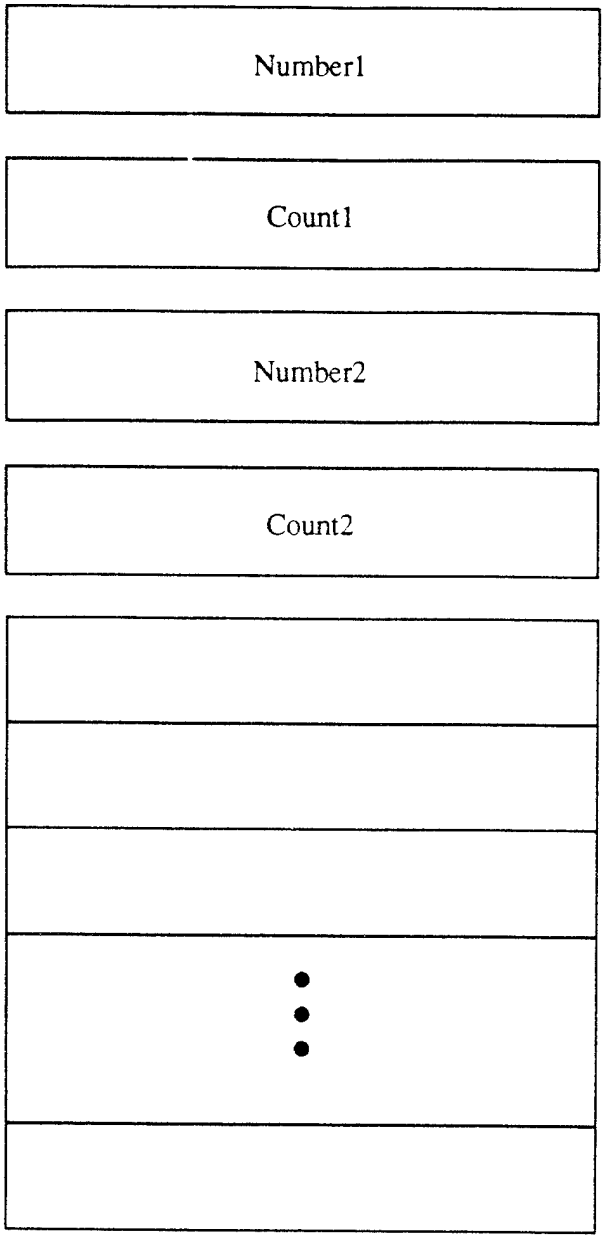
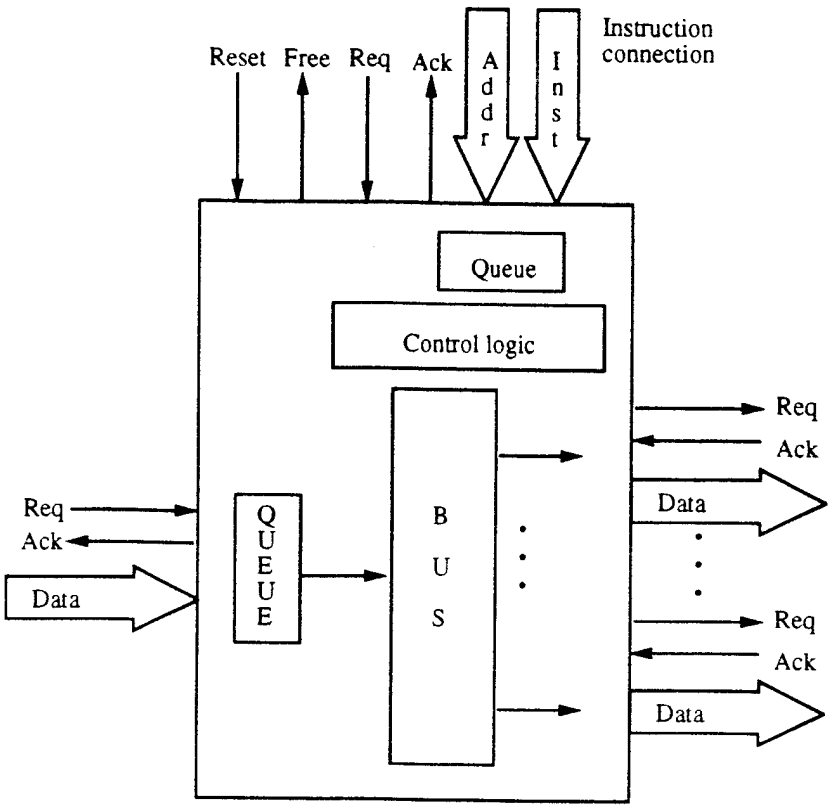
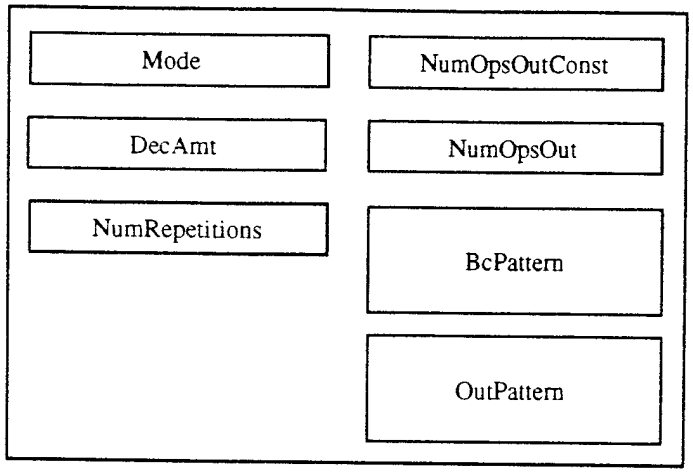


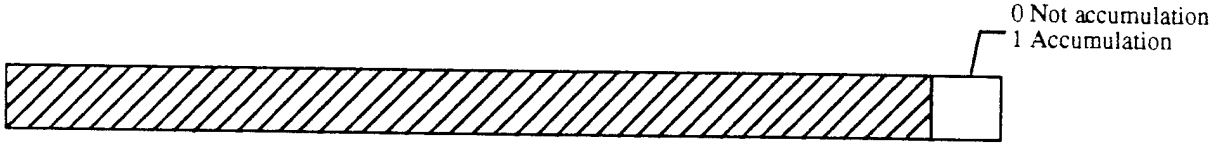
Fig. 1.7. Pattern registers.



(a) Block Diagram.

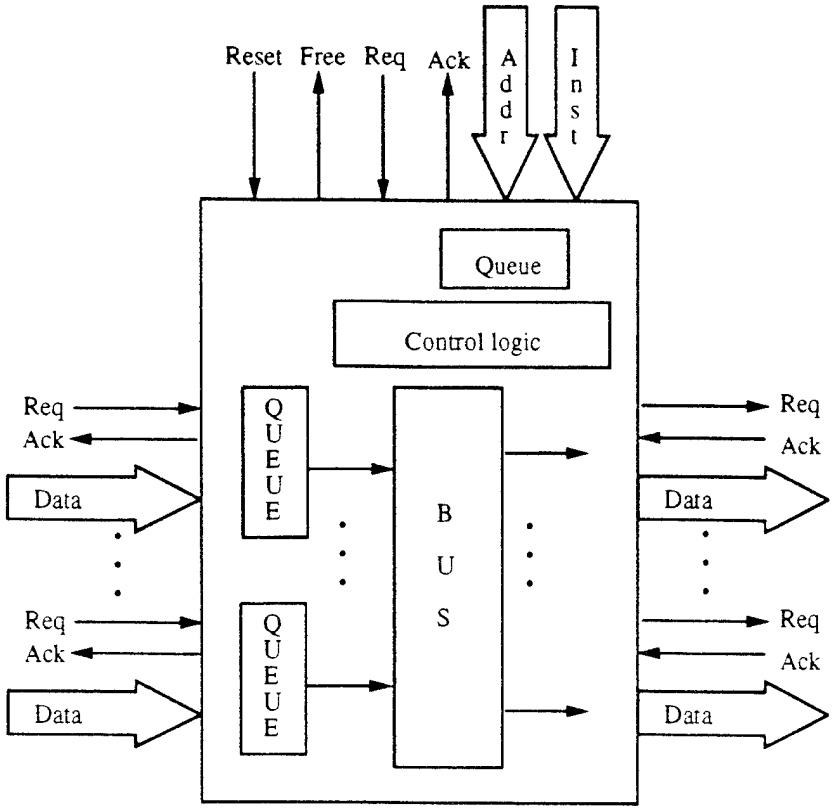


(b) Programmable registers.

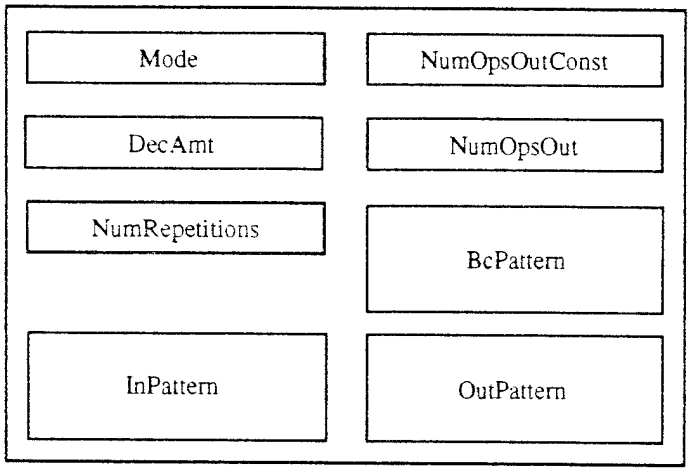


(c) Mode register.

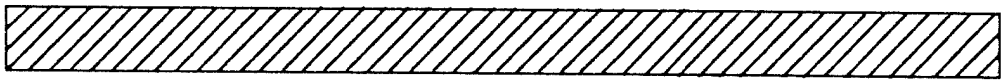
Fig. 1.8. F component



(a) Block Diagram.



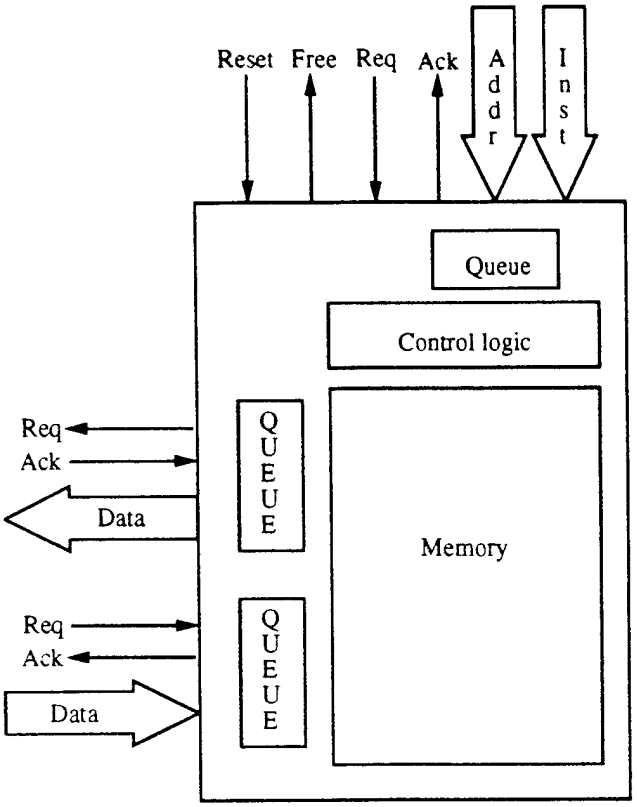
(b) Programmable registers.



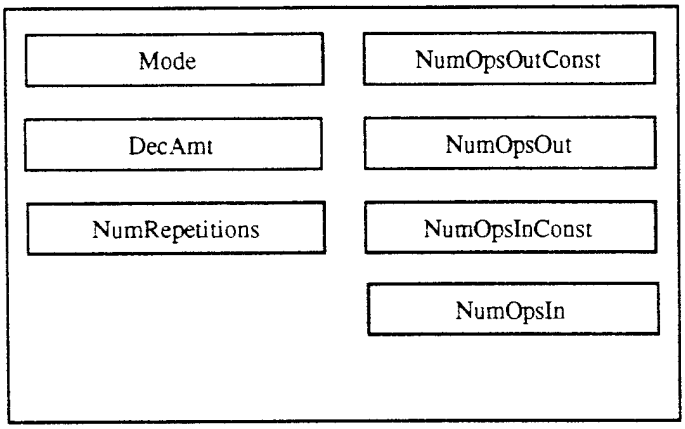
Note: Currently the entire mode register is not used

(c) Mode register.

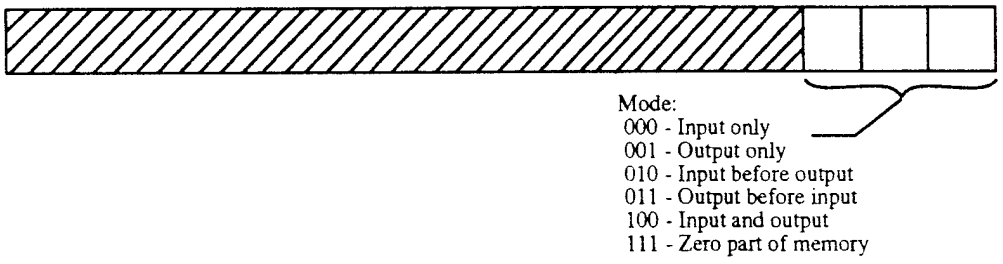
Fig. 1.9. L component



(a) Block Diagram.



(b) Programmable registers.



(c) Mode register.

Fig. 1.10. R component

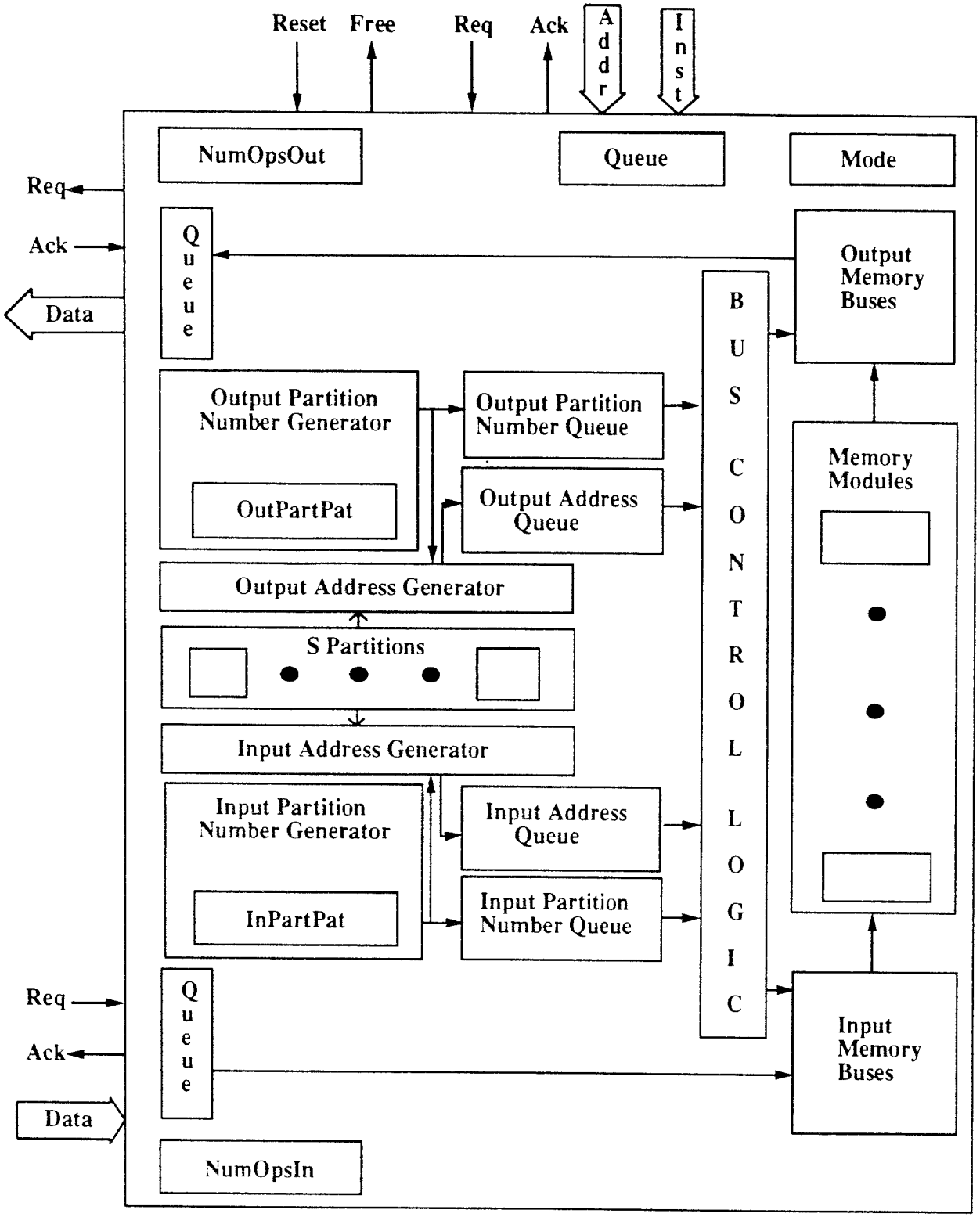
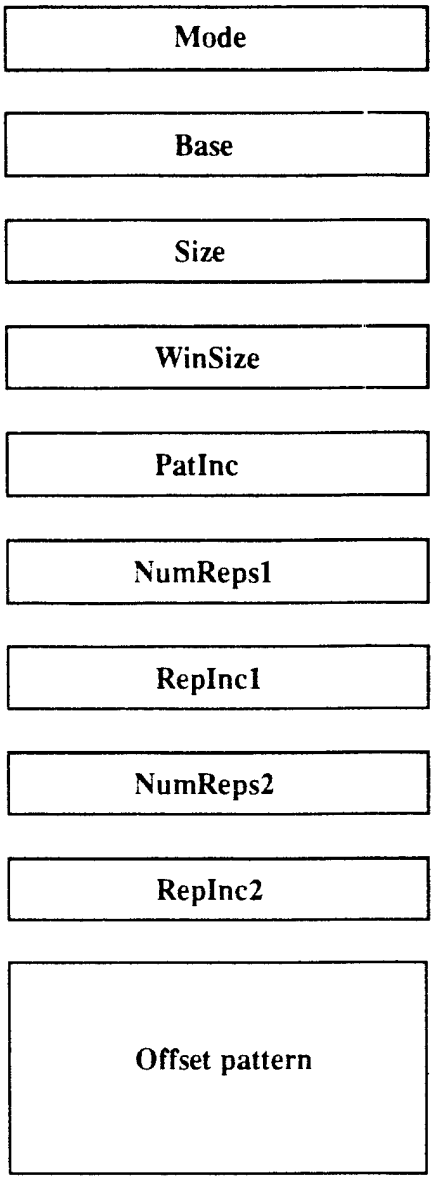
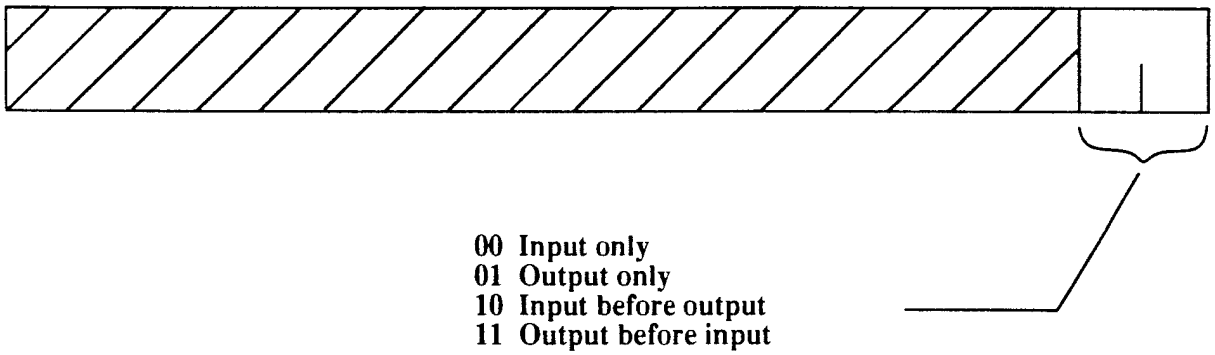


Fig. 1.11. S -Component Block Diagram



(a) Registers



(b) Mode register format

Fig. 1.12. Registers in an S partition.

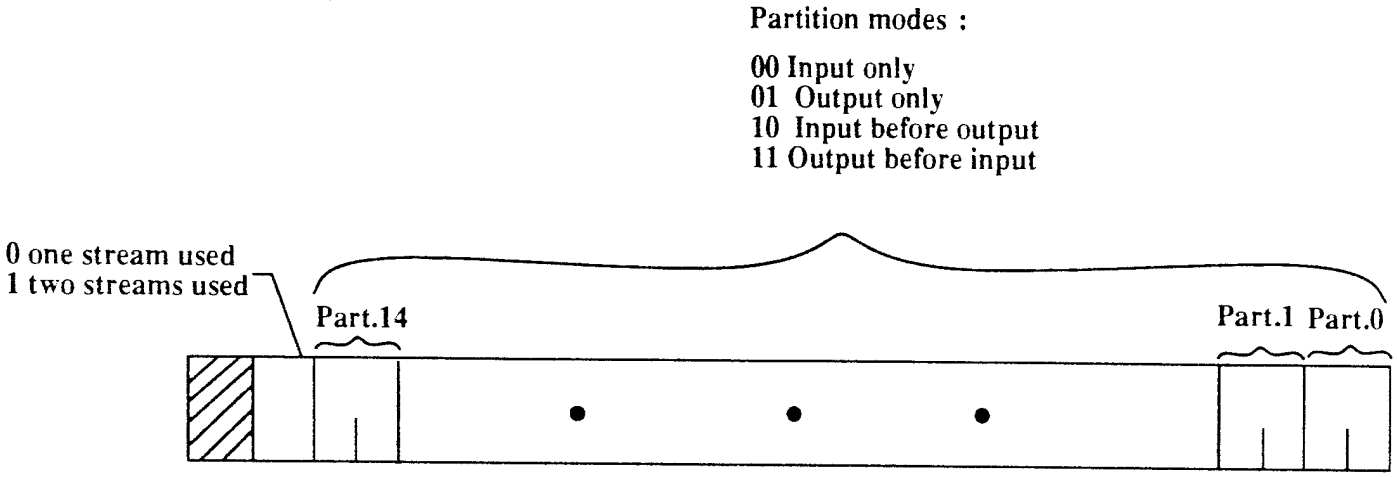


Fig. 1.13. S component mode instruction format.

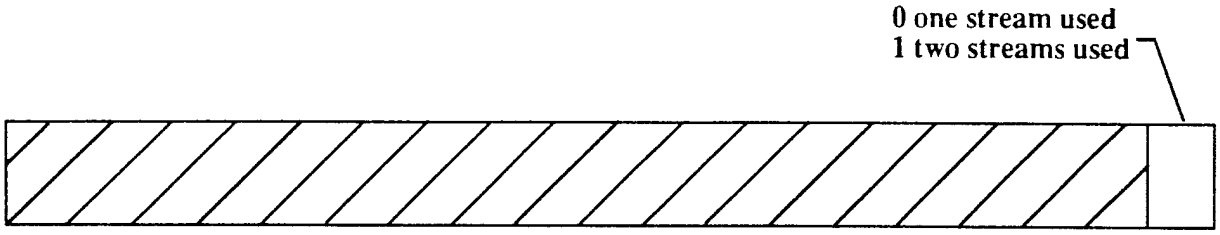


Fig. 1.14. S component mode register.

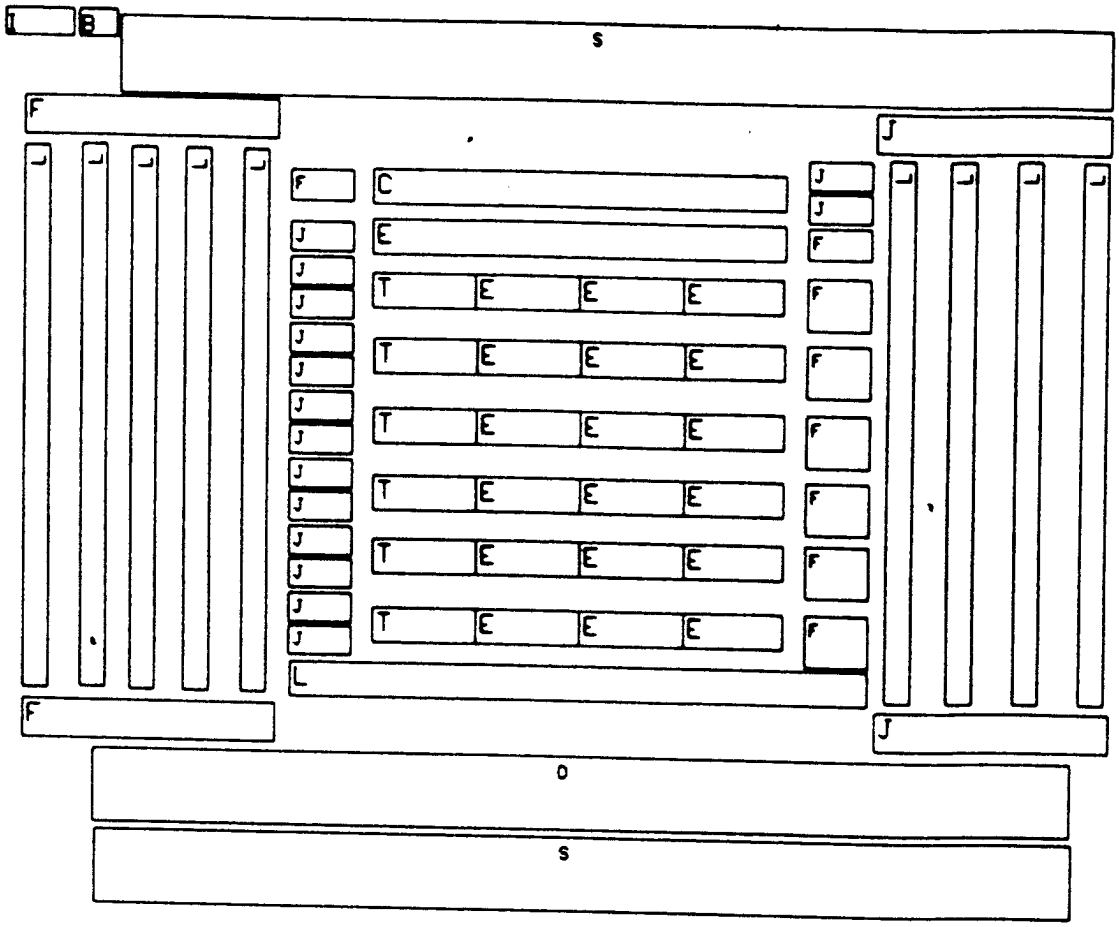


Fig 1.14 Example MCAP for performing matrix operations.

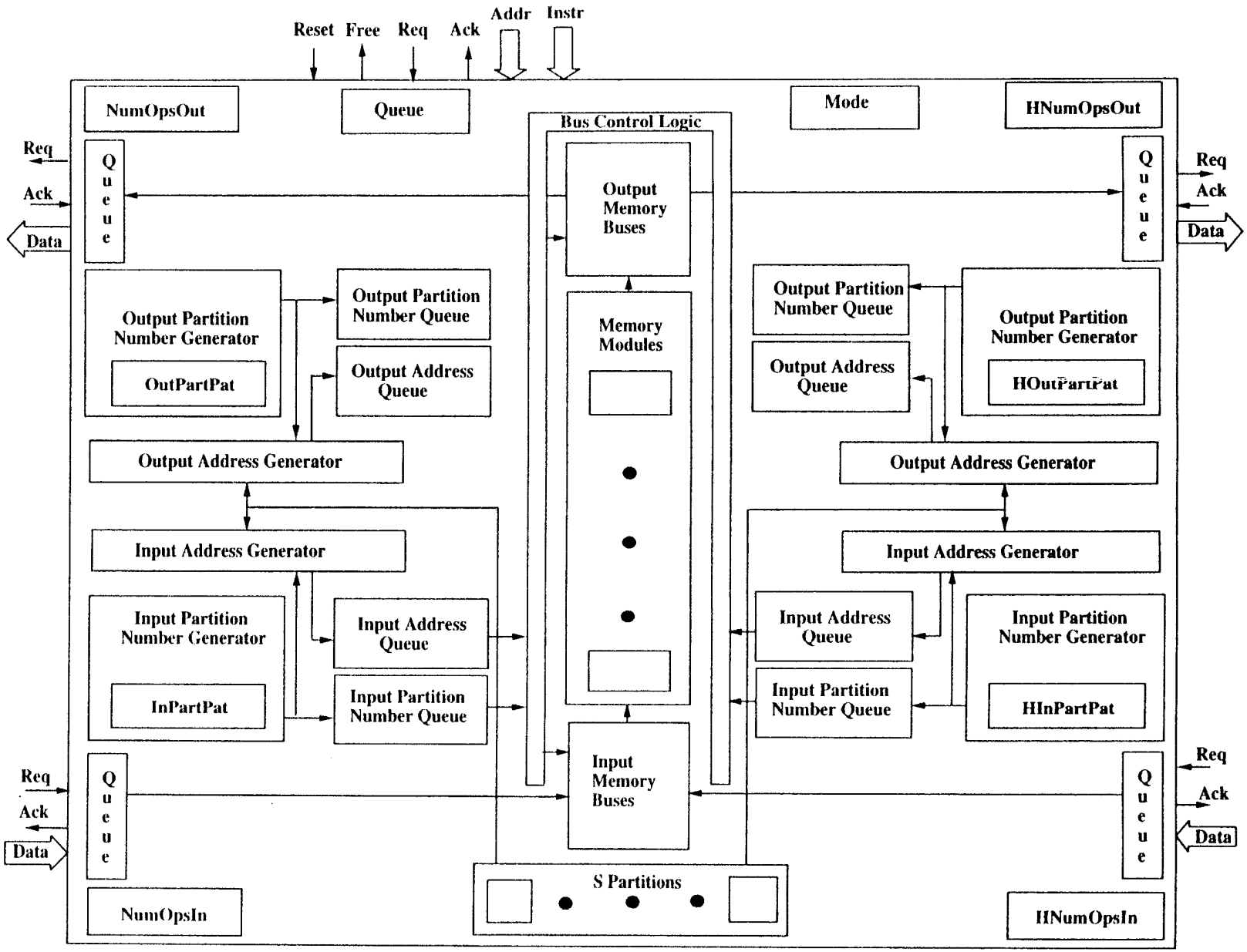
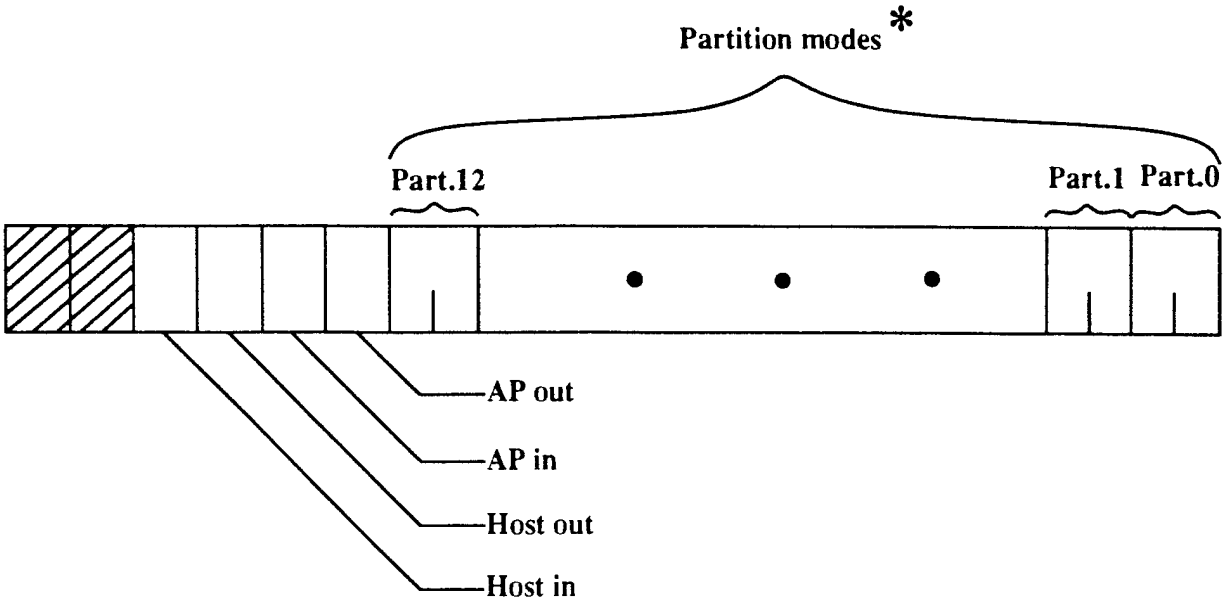
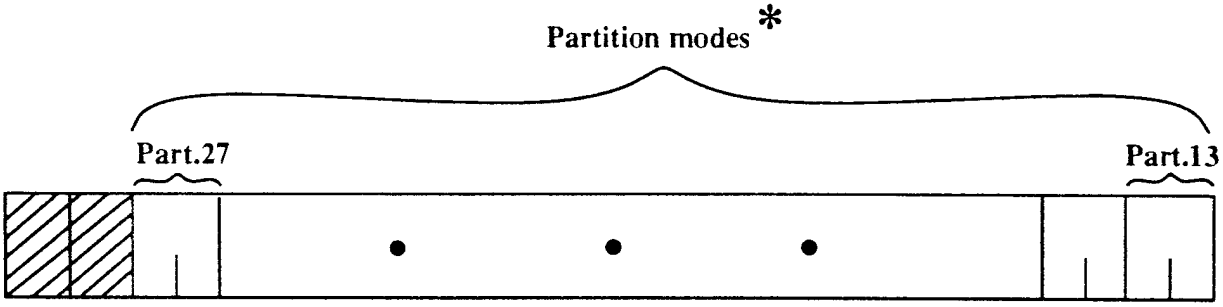


Fig. 1.15. D-Component Block Diagram



(a) DMOD instruction



(b) DXMD instruction

* Partition modes :

00 Input only
01 Output only
10 Input before output
11 Output before input

Fig. 1.16. D component mode instruction formats.

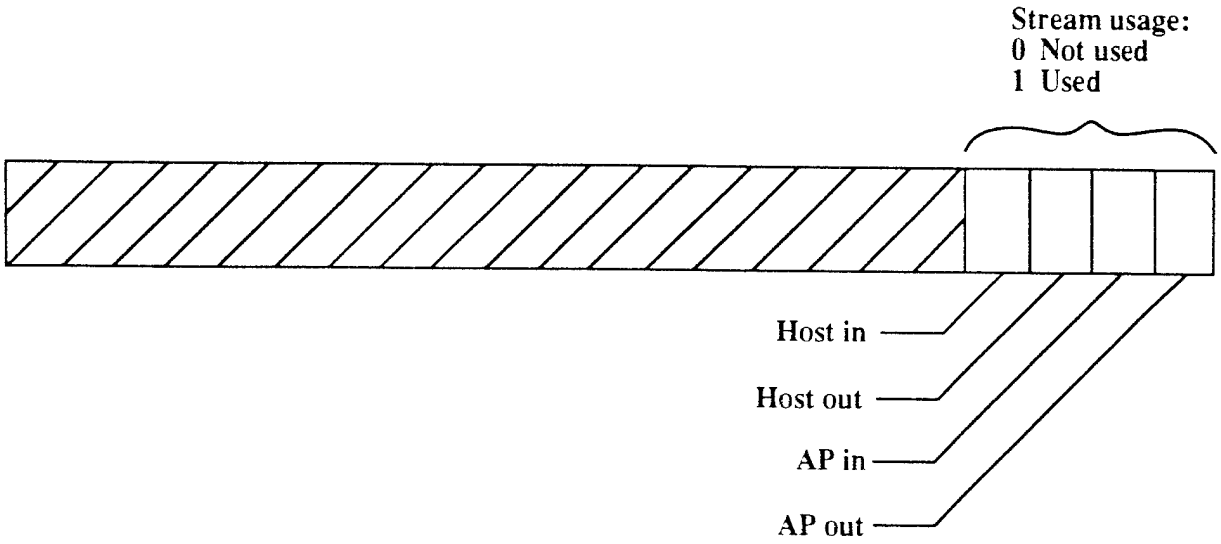


Fig. 1.17. D component mode register.

ATTACHMENT B

CHAPTER 2
PROGRAMS AND FILES ASSOCIATED WITH SIMARC SIMULATION PACKAGE

The programs in the simulate architecture (SIMARC) system of programs communicate through a collection of files. The relationship between the programs and the files they create and access is summarized in Fig. 2.1. The text editor is not a member of the simulate architecture program set, but may be any text editor capable of producing an ASCII text file. The editor currently used is the DOS program *EDIT*.

The files used by the system may be given any legitimate DOS filenames, but must have the extensions shown in the figure. These files are classified as follows:

Architecture file (.SAR) - contains a complete description of an architecture

Architecture error file (.SAE) - contains connection errors made while generating an architecture

Information file (.SIF) - contains a textual description of the architecture.

Program file (.SAS) - contains a source program in the form needed by the assembler.

List file (.SLT) - contains a list of the errors produced during an assembly.

Load file (.SLD) - contains a program in the form needed by the simulator.

Result file (.SRT) - contains the results produced by a simulation.

Irregular intervals file (.SIR) - specifies the time at which results are to be taken when the result intervals are irregular.

Fractional result file (.SFT) - contains the same information as an .SRT file except that the results are given as percentages.

Summary file (.SUM) - contains automatically updated summary information from succesive tests.

The programs are:

EDITOR - for creating a new architecture or changing an existing architecture.

ARCHCHK -	checks an architecture's connections for errors. It is automatically executed each time a .SAR file is created or updated.
AS -	for assembling a source program into a load form that can be used by the simulator (SIMULATE).
SIMULATE -	for simulating a given program on a given architecture and accumulating results of the simulation.
DISPLAY -	for displaying the results of a simulation on the monitor.
SIMSUMM -	for producing summary information as well as storing the results as percentages

Each type of file is described in a subsection given below. Each subsection defines the content and format of one of the file types.

2.1 Architecture File

An architecture (.SAR) file completely describes the architecture of an MCAP and provides the graphics information needed to display the architecture and mnemonics required by the assembler. An MCAP consists of a standard set of connections and components in which each end of each connection is attached to exactly one component. An MCAP is completely defined by specifying the

- Connection type for each connection and the component that is attached to each end of each connection.
- Component type and the attributes of each component.

An architecture file is an ASCII file that is divided into two parts. The two parts are separated by a string consisting entirely of one or more asterisks. The first part is for storing connection information that can be easily used to modify an existing architecture. The second part contains the information needed by the assembler and simulator.

The second part of the architecture file is broken into fields, one field for each component in the MCAP. The first part of a field contains the information that is common to all components and is summarized in Fig. 2.2. The remaining information in a field is determined by the component's type. Figures 2.3 through 2.9 summarize this information for each type. In all of these figures the data type for each entry or subfield is given in parentheses. Except for the type, mnemonic and component number, the data corresponding to each major item in these figures is on a single line in the architecture file, even if it is a single number.

2.2 Information File

An information (.SIF) file simply gives an easily readable description of an architecture. It has the same file name as the architecture file that contains the architecture it describes and consists of a listing of the components and their attributes.

2.3 Architecture Error File

An architecture errors (.SAE) file contains descriptions of the connection errors made while using EDITOR to create an architecture and a summary of all of the architecture's connections. It has the same file name as the architecture file containing the errors.

2.4 Program Source File

A program source (.SAS) file is an ASCII file in which each line is blank, a remarks line, or an assembler language directive or instruction for an MCAP. At present, there are three directives. They are given in Fig. 2.10. In an EQU statement the symbol *Name* is assigned the value *Constant*. *Remarks* is optional and can be any string of text. If *Remarks* is not present, then the semicolon is optional. Also, a line may begin with a semicolon, in which case the remainder of the line can be any text. PROC and ENDP mark where instructions are placed that make up procedures. An EQU statement can not be placed inside the directive pair PROC...ENDP and an instruction can not be placed outside them.

There are two types of instructions, internal instructions and external instructions. Internal instructions are those that are executed entirely within the instruction component and external instructions are those that are distributed to and executed by other non-memory components. Instructions have the format:

Label: Mnemonic Component Operand,...,Operand ;Remarks

where *Label*, the colon, the semicolon, and *Remarks* are optional, but if *Label* is present then the colon must be present and if *Remarks* is present then the semicolon must be present. *Component* appears in external instructions only. *Mnemonic* and *Component* must be followed by at least one space character. *Operands* must be separated by a comma or at least one space character.

Label is a string for identifying the instruction. *Mnemonic* indicates the instruction type and *Component* is a mnemonic that indicates the component that ultimately decodes and executes the instruction. There may be none, one or more than one *Operand* depending on the type of instruction.

The internal instructions are summarized in Fig. 2.11. They include instructions for

implementing subprograms and loops, manipulating the data in the instruction component's registers, and halting the computer. Also, a no–operation instruction is included.

The external instructions are summarized in Fig. 2.12. They are for putting values into the various registers in the programmable components (i.e., the E, T, C, J, F, L, R, S and D components). These registers determine the activities within their respective components. During the execution of an algorithm, the values in these registers designate the number of operands out, number of operands in, mode, and immediate operands; input and output patterns for routing; and memory partitioning and access patterns for memory controllers. The value put in one of these registers may be immediate (i.e., the value indicated by the operand) or be in the instruction component's register whose ID is indicated by the operand. A register ID begins with an asterisk and ends with up to five digits.

2.5 List File

A list (.SLT) file consists of a simply listing of the errors produced during the assembly process. It has the same file name as the source program file that is assembled. The format for the list file consists of the error messages for each line containing an error followed by the line itself.

2.6 Load Files

A load (.SLD) file is an ASCII file that is in the program format needed by the simulator. As indicated in Fig. 2.13, the first line corresponding to an instruction begins with the instruction's opcode, the type of component that is to receive and execute the instruction, and the component's number. The opcodes for the instructions are given in Fig. 2.14.

This information is followed by a sequence of numbers that are the operands. If an instruction pertains to a particular partition, then the first of these numbers identifies the partition. The instructions that refer to a particular partition are SOSP, SPNI, SPBS, SWIS, DOSP, DPNI DPBS, and DWIS (see Fig. 2.12 in Sec. 4). If an instruction includes an indeterminate number of operands, then the number preceding the indeterminate set of operands indicates the number of operands in the set. For JSIP, FSOP, LSIP, LSOP, SIPP, SOPP, DIPP, DOPP, DPPI and DPPO two sets of numbers are used to specify a pattern. In each set there is a sequence of items of indeterminate length preceded by the number of items in a sequence, which is, in turn preceded by the total number of times the items of the sequence that are used before the other sequence is used. Usage of the sequences alternate with the first sequence being employed first. FSBP and LSBP include only one sequence but in order to make them consistent with the other pattern instructions, two 0's are appended to them. Fig. 2.15 gives a complete description of the operands for the various instructions.

Note that for the SPNI and DPNI instruction the last four operands or the last two

operands may be omitted in the source code. If omitted, these operands are set to 0 in the load code.

In Sec. 4 it was indicated that an operand may be immediate or the ID of an instruction component register. If an operand is a register ID, then the corresponding operand in the load file will begin with an asterisk and end with up to five digits. The five digits, of course. identify the register.

2.7 Result File

The result (.SRT) file is an ASCII file that contains the results of a program simulation. As indicated by Fig. 2.16(a), a result file consists of lines with the first line containing the filename of the tested architecture. The second line indicates the number of components in the architecture. The third line indicates whether regular or irregular intervals were used to collect the results. the fourth gives the total area and the fifth provides the column headings. Column headings are included to make the file more readable. The remainder of the file consists of a field for each interval for which results were recordered.

As shown in Fig. 2.16(b), each field begins with the system time at which the data was taken. In the remainder of a field is one or more lines for each component. Fig. 2.16(c) gives the format of the lines. Each line corresponds to a component or, for I, R, S and D components. a set of logic with a component. Each line begins with the component's mnemonic followed by the area of the component or set of logic for wich the data was collected. The area is followed by the current accumulated times the component or set of logic has spent in each state since the beginning of the program and the maximum number of entries in the component's instruction and data queues since the beginning of the program. Entries that do not apply to a component or set of logic are filled with 0's.

The possible states that a component can be in are defined below. A component is in exactly one of these states at a time. Some types of components can assume any one of these states at a given time, while others can assume only some of the states. The component types that can take on a state are given in parentheses.

BUSY (all component types) - the component is actively performing one of its functions.

WAIT (all component types) - the component is waiting for its output to be taken or, for the I component, a flag to be set.

IDLE (I,E,T,C,J,F,L,R,S,D) - the component is waiting for input.

FREE (all component types) - the component is completely inactive.

DIST (E,T,C,J,F,L,R,S,D) - the component is distributing instructions to its registers.

For the S and D components, only the input stream logic and output stream logic can take on all five states, the address generator, buses and memory modules can be only FREE or BUSY. Also, depending on the component type, the state time data may be followed by the maximum numbers of entries in the component's instruction and data queues. The component types I, B, E, T, C, J, F, L, R, S and D have instruction queues and the component types E, T, C, J, F, L, R, S and D have input data queues. Component types J and L have multiple input data queues and only the maximum number of entries taken over all queues is recorded. Component types R and S also have output data queues and D components have two input queues and two output queues. If a component does not have an instruction or data queue, then a 0 is entered as the corresponding data.

Results can be saved at regular or irregular intervals. Regular intervals are used to specify saving results every multiple of a value. For example, if a regular interval of 1000 is specified, results will be saved at 1000, 2000, 3000, etc. up to the last multiple of 1000. In addition, the results are saved at the end of the simulation if they weren't previously saved as a multiple of the regular interval. If intervals other than a multiple of some specified value are required, irregular intervals must be used.

Irregular intervals specify the exact point(s) at which results are to be saved. As many as 100 of these values may be entered for any simulation. In addition, the results are saved at the end of the simulation unless this would be a duplication of the last interval specified.

2.8 Irregular Intervals File

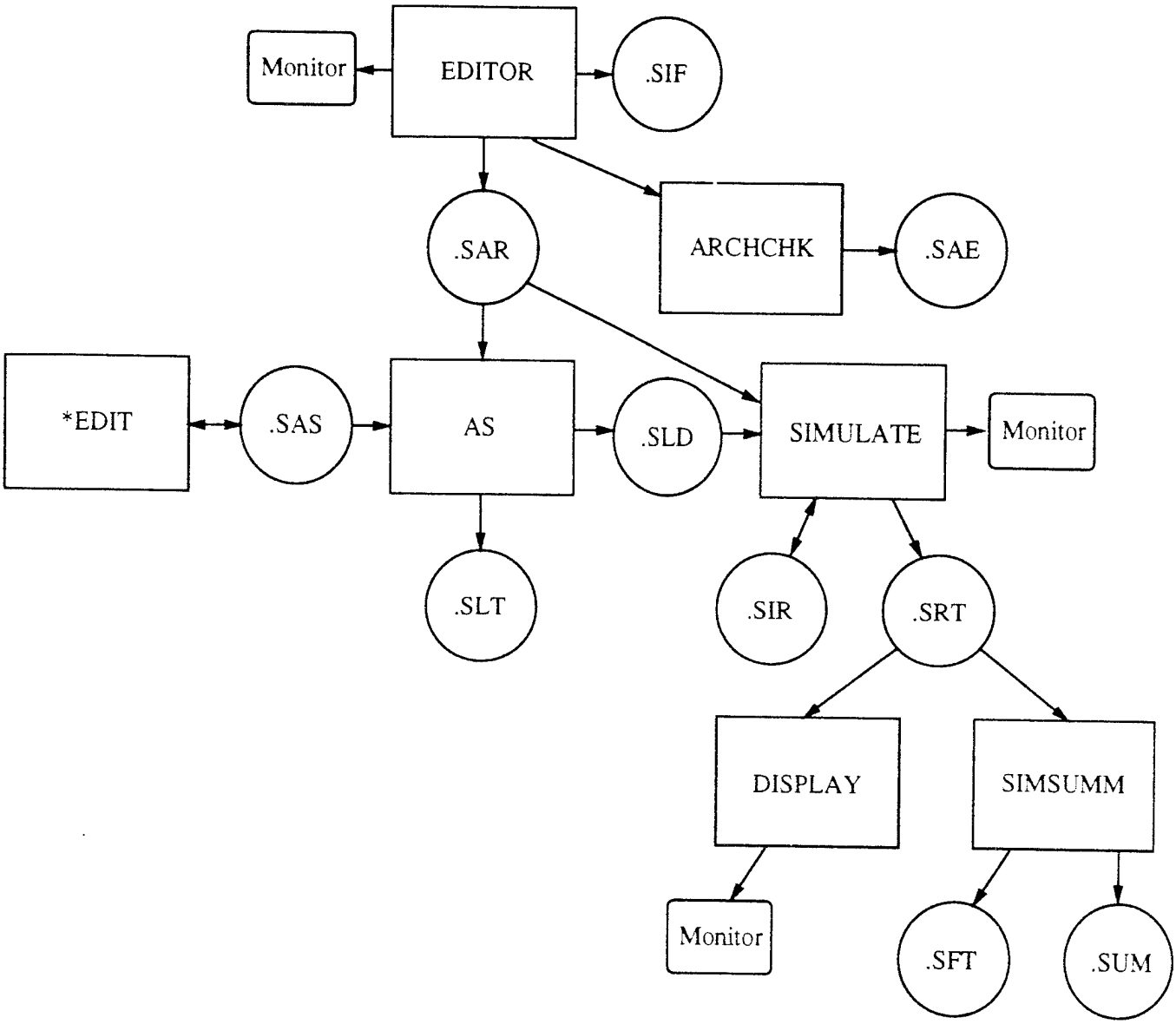
When the user chooses to record results at irregular intervals, the system times for recording the results may be entered through the keyboard or taken from an irregular intervals (.SIR) file, whose format is shown in Fig. 2.17. The file simply consists of the times followed by a 0, which serves as a terminator. If the user chooses to enter the times through the keyboard, then he or she may also choose to store the times in an .SIR file so that they can be used later without re-entering them.

2.9 Fractional Result File

A fractional result (.SFT) file contains the same information and has the same format as the result (.SRT) file, except that the times spent in each state are replaced by percentages of times spent in each state. For example, if the system time were 1500 and the time spent in the BUSY state were 900, then 60 would replace 900 in the result data.

2.10 Summary File

A summary (.SUM) file is for storing BUSY and sustainable speed data from a sequence of tests. Each time the file is accessed the user is asked to enter a comment through the keyboard and the data needed to compute the BUSY and speed information is extracted from the specified result (.SRT) file. The BUSY and speed information is then appended to the .SUM file using the format given in Fig. 2.18.



* Although the SIMARC package uses the DOS program EDIT, any text editor capable of producing an ASCII file, could be used to produce a .SAS file.

Fig. 2.1 Relationship between the SIMARC package's programs and files.

*Type (a single character I, B, M, E, T, C, J, F, L, R, S or D)

*Mnemonic (string of four letters and digits)

*Component number (positive integer)

Number of rectangles used to display component (positive integer)

**An array of quadruplets, each specifying a rectangle as follows:

Left (non-negative integer)

Top (non-negative integer)

Right (non-negative integer)

Bottom (non-negative integer)

A pair specifying position of component's displayed text

X position (non-negative integer)

Y position (non-negative integer)

Text size (non-negative integer)

Text orientation (positive integer)

Execution time (non-negative integer)

*Type must be the first character on a line and Type, Mnemonic and Component ID constitute a line.
**Size of array is the number of rectangles.

Fig. 2.2 Attributes common to all components

Instruction queue size (positive integer)

Memory Time (positive integer)

Area (positive integer)

Memory Area (positive integer)

(a) I component

Instruction queue size (positive integer)

Area (positive integer)

(b) B component

Fig. 2.3 Additional features for the I and B components.

Alternate execution time (non-negative integer)

Distribution time (positive integer)

*Number of stages in accumulation pipeline (non-negative integer)

Input connection (positive integer)

Output connection (positive integer)

Instruction queue size (positive integer)

Data queue size (positive integer)

Area (positive integer)

**Additional input connection (positive integer)

*0 if component is not part of an accumulation pipeline
**Included in T or C component only.

Fig. 2.4 Additional attributes for processor components.

Distribution time (positive integer)

*Number of stages in accumulation pipeline (non-negative integer)

Number of input connections (positive integer)

**Array of input connection numbers (positive integers)

Output connection number (positive integer)

Instruction queue size (positive integer)

Data queue size (positive integer)

Area (positive integer)

*0 if component is not part of an accumulation pipeline
**Size of array is the number of input connections

Fig. 2.5 Additional attributes for the J component.

Distribution time (positive integer)

*Number of stages in accumulation pipeline (non-negative integer)

Input connection number (positive integer)

Number of output connections (positive integer)

**Array of output connection numbers (positive integers)

Instruction queue size (positive integer)

Data queue size (positive integer)

Area (positive integer)

*0 if component is not part of an accumulation pipeline
**Size of array is the number of output connections

Fig. 2.6 Additional attributes for an F component.

Distribution time (positive integer)

Number of input connections (positive integer)

*Array of input connection numbers (positive integers)

Number of output connections (positive integer)

**Array of output connection numbers (positive integers)

Instruction queue size (positive integer)

Data queue size (positive integer)

Area (positive integer)

*Size of array is the number of input connections
**Size of array is the number of output connections

Fig. 2.7 Additional attributes for an L component.

Distribution time (positive integer)

Input connection number (positive integer)

Output connection number (positive integer)

Capacity of memory (positive integer)

Instruction queue size (positive integer)

Data queue size (positive integer)

Area (positive integer)

Fig. 2.8 Additional attributes for the R component.

Input execution time (non negative nteger)

Distribution time (positive integer)

Memory module capacity (positive integer)

Input connection number (positive integer)

Output connection number (positive integer)

Number of memory modules per bank (positive integer)

Number of memory banks (positive integer)

Address time (positive integer)

Memory time (positive integer)

Memory bus time (positive integer)

Number of output memory buses (positive integer)

Number of input memory buses (positive integer)

Instruction queue size (positive integer)

Data queue size (positive integer)

Address queue size (positive integer)

Stream area for each stream (positive integer)

Memory area for each memory module (positive integer)

Bus area for each bus (positive integer)

Address generator area for each generator (positive integer)

*Host connection number in (positive integer)

*Host connection number out (positive integer)

*These attributes are for D components only.

Fig. 2.9 Additional attributes for the S and D components.

Name EQU *Constant*

(a) EQU directive.

label: mnemonic operand,..., operand

label: mnemonic operand,..., operand

⋮

label: mnemonic operand,..., operand

(b) PROC .....ENDP directives.

Fig. 2.10 Directive summary.

Format		Description
CALL	*Label*	Subroutine call to *Label*
RTRN		Subroutine return
LOOP	*Operand, Label*	Repetition count = *Operand*
		Branch address = *Label*
NOOP		No operation
HALT		Terminates the program
WAIT	*Operand*	0 - waits for all components to be free
		1 - waits for comparator to be free
MOVE	*R, Operand*	(R) <- *Operand*
ADDR	*R, Operand*	(R) <- (R) + *Operand*
SUBR	*R, Operand*	(R) <- (R) - *Operand*
MULR	*R, Operand*	(R) <- (R) * *Operand*
DIVR	*R, Operand*	(R) <- Quotient (R)/ *Operand*
		(R+1) <- Remainder (R)/ *Operand*
NEGR	*R*	(R) <- - (R)
BRAN	*Label*	Branch to *Label*
BREQ	*R, Operand, Label*	Branch to *Label* if (R) = *Operand*
BRNE	*R, Operand, Label*	Branch to *Label* if (R) <> *Operand*
BRGT	*R, Operand, Label*	Branch to *Label* if (R) > *Operand*
BRGE	*R, Operand, Label*	Branch to *Label* if (R) >= *Operand*
BRLT	*R, Operand, Label*	Branch to *Label* if (R) < *Operand*
BRLE	*R, Operand, Label*	Branch to *Label* if (R) <= *Operand*
STOP	*Operand*	Sets a breakpoint at time specified by *Operand*
RSET		Resets all components

Legend:

Label	- instruction label
Operand	- a constant or register ID
R	- an instruction component register number

Fig. 2.11 Internal instruction summary.

Format		Description
(Type)IMM	*Component Oprd*	Immediate operand =*Oprd*
(Type)NOO	*Component Oprd*	Number of operands out = *Oprd*
(Type)NOI	*Component Oprd*	Number of operands in = *Oprd*
(Type)MOD	*Component Oprd Component**	Mode = *Oprd*
(Type)REP	*Component Oprd*	Number of repetitions = *Oprd*
(Type)DEC	*Component Oprd*	Decrement amount =*Oprd*
JSIP	*Component #Oprd In ... In #Oprd In ... In*	Set input pattern
FSOP	*Component #Oprd Out ... Out #Oprd Out ... Out*	Set output pattern
FSBP	*Component Out ... Out*	Set broadcast pattern
LSIP	*Component #Oprd In ... In*	Set input pattern
LSOP	*Component #Oprd Out ... Out*	Set output patern
LSBP	*Component Out ... Out*	Set broadcast pattern
SIPP	*Component #Oprd Part ... Part #Oprd Part ... Part*	Set input partition pattern
SOPP	*Component #Oprd Part ... Part #Oprd Part ... Part*	Set output partition pattern
SOSP	*Component Part #Oprd Os ... Os #Oprd Os ... Os*	Set partition offset pattern
SPNI	*Component Part PatInc NR1 RI1 NR2 RI2* **	Set repetitions and increments
SPBS	*Component Part PartBase PartSize*	Define partition
SWIS	*Component Part WinSize*	Define window
DXMD	*Component Oprd*	Extra mode = *Oprd*
DIPP	*Component #Oprd Part ... Part #Oprd Part ... Part*	Set input partition pattern
DOPP	*Component #Oprd Part ... Part #Oprd Part ... Part*	Set output partition pattern
DOSP	*Component Part #Oprd Os ... Os #Oprd Os ... Os*	Set partition offset pattern
DPNI	*Component Part PatInc NR1 RI1 NR2 RI2* **	Set repetitions and increments
DPBS	*Component Part PartBase PartSize*	Define partition
DWIS	*Component Part WinSize*	Define window
DPPI	*Component #Oprd Part ... Part*	Set partition pattern in
DPPO	*Component #Oprd Part ... Part*	Set partition pattern out
DHNO	*Component Oprd*	Number of host operands out = *Oprd*
DHNI	*Component Oprd*	Number of host operands in = *Oprd*

* For T and C component, *Component* indicates the single input or, for two inputs, the variable input; otherwise, this operand is not present.

** The pair NR2, RI2 or both it and the pair NR1,RI1 may be omitted, in which case each omitted operand is set to 0.

Note: *#Oprd* may not be present, but must appear twice or not at all. If not present, then the entire list is continually cycled through.

Legend:

Oprd	- integer or, if preceeded by an asterisk, a register ID
Component	- mnemonic of component to be programmed
In	- component mnemonic of data source
Out	- component mnemonic of data destination or & if broadcasting
Part	- partition number or register ID
PatInc	- pattern increment or register ID
NR1,NR2	- number of repetitions or register ID
RI1,RI2	- repetition increment or register ID
PartBase	- partition base address or register ID
PartSize	- partition size or register ID
Os	- offset in a partition's offset pattern or register ID
WinSize	- window size or register ID

Fig. 2.12 External instruction summary.

Opcode *Component type* *Component ID* *Sequence of numbers**

* In the sequence of numbers some of the numbers may be preceded by an asterisk, in which case the number is a register ID.

Fig. 2.13 General format of a load file instruction.

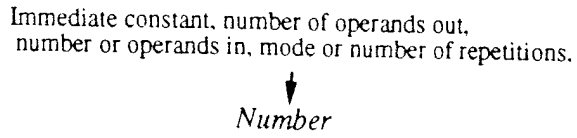
Opcode	Mnemonic
1	CALL
2	RTRN
3	LOOP
4	NOOP
5	HALT
6	WAIT
7	MOVE
8	ADDR
9	SUBR
10	MULR
11	DIVR
12	NEGR
13	BRAN
14	BREQ
15	BRNE
16	BRGT
17	BRGE
18	BRLT
19	BRLE
20	STOP
21	RSET

(a) Internal instructions

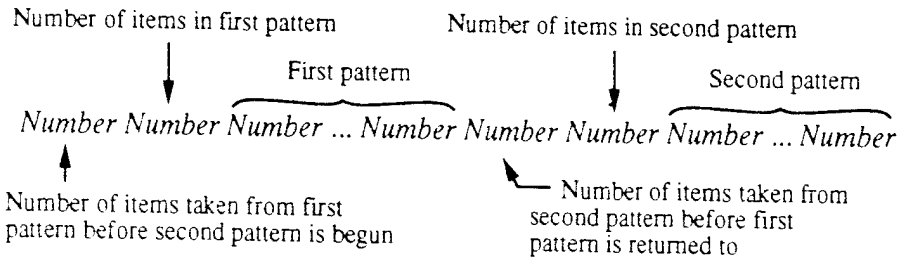
Opcode	Mnemonic
50	(Type)IMM
51	(Type)NOO
52	(Type)NOI
53	(Type)MOD
54	JSIP
55	FSOP
56	FSBP
57	LSIP
58	LSOP
59	LSBP
60	SIPP
61	SOPP
62	SOSP
63	SPNI
64	SPBS
65	DIPP
66	DOPP
67	DOSP
68	DPNI
69	DPBS
70	DWIS
71	DPPI
72	DPPO
73	(Type)REP
74	(Type)DEC
75	DHNO
76	DHNI
77	SWIS
78	DXMD

(b) External instructions.

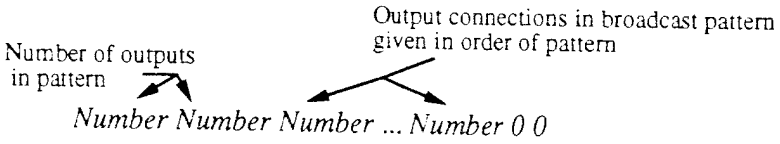
Fig. 2.14 Opcode assignments.



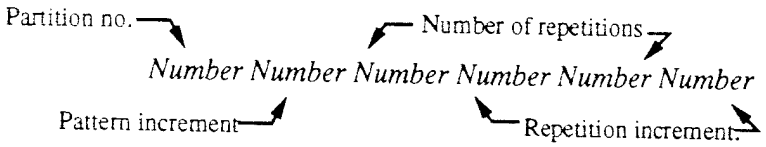
(a) IMM, NOO, NOI, MOD, REP, DEC, DHNO or DHNI instruction.



(b) JSIP, FSOP, LSIP, LSOP, SIPP, SOPP, DIPP, DOPP, DPPI or DPPO instruction.



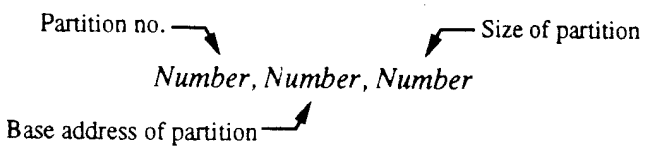
(c) FSBP or LSBP instruction.



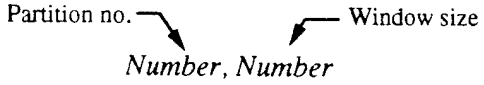
(d) SPNI or DPNI instruction.

Note: Some *Number* fields may be followed by an asterisk, in which case the *Number* is a register ID and the contents of the register are to be used when the instruction is executed.

Fig. 15 Formats of the Sequence of Numbers field given in Fig. 14



(e) SPBS or DPBS instruction.



(f) SWIS and DWIS instruction.

Fig. 2.15 (Continued)

Architecture filename

Number of components in system

Type of interval (Regular Interval or Irregular Interval)

Total area

Column headings

Results field

.

.

.

Results field

(a) Overall format.

System time

Line of data

.

.

.

Line of data

(b) Results field format

+Component Area BUSY time WAIT time IDLE time FREE time DIST time *IQ *DQ

+ Entries that are not applicable to a component are filled with 0's.

* Maximum number of entries in the instruction or data queue up to the present time. For multiple inputs, the maximum is taken over all data queues.

Note: For the I component the results occupy two lines, one for the decode logic and one for the memory. For R components, the results occupy two lines , one for the input stream and one for the output stream. For S and D components, the results occupy one line for each data stream, each address generator, each bus and each memory module.

(c) Line of data format.

Fig. 2.16 Result file format.

System time at end of interval

.

.

.

System time at end of interval

0

Note: The final zero serves as a terminator.

Fig. 2.17 Irregular interval file format.

Comment

Percent BUSY for E and T components: *Percentage*

Average sustainable speed: *Value* MFLOPS

Fig. 2.18 Format of a summary (.SUM) file entry.

ATTACHMENT C

DESIGN CONSIDERATIONS FOR IMPLEMENTING A MODULARLY CONFIGURED ATTACHED PROCESSOR IN A MULTI CHIP MODULE

J. Sanjay Singh, Buck W. Gremel, Vijay P. Singh, and Glenn A. Gibson
Electrical and Computer Engineering Department
The University of Texas at El Paso
El Paso, Texas 79968-0523

Abstract

Implementation of a novel modularly configured attached processor (MCAP) architecture was evaluated using 1 μm CMOS logic on an MCM-D. The transistor count was approximately ten million transistors, distributed on twenty-five chip dice. Delay, area, and power calculations were performed using the SUSPENS model. Rent's rule was found to be not applicable. Speed was calculated to be in the 100 MHz range. The module foot print was found to be 90 cm^2. Power dissipation per unit area was low enough to allow air cooling.

1 Introduction

Attached processors [1], [2], [3] are commonly used for the purpose of very quickly executing most of the system's computational tasks. In such an organization, "the host is a program manager which handles all I/O, code compiling, and operating system functions, while the attached processor concentrates on arithmetic computation with data supplied by the host" [1].

In addition to quick execution, it is also desirable to execute as broad a set of algorithms as possible in order to create a more generally applicable processor. Thus, the underlying goal of the designer is to efficiently utilize the hardware for as broad a set of algorithms as possible. However, for most current designs, the average sustainable execution rates have been found to be only 5% to 20% of their peak rates. For example, the sustainable rates for the Cray X-MP with four processors may be as low as 5% for some algorithms [4]. Although some of the lost efficiency is necessitated by the algorithms, much of it is due to memory accessing and contention for shared resources in general, including internal buses.

In this article we describe the implementation of a novel modularly configured architecture wherein utilization of each processor is greatly enhanced through: (1) closely matching their architectures to the set of algorithms they are to execute, (2) overlapping of processing and memory accessing by using memory prefetching, (3) minimizing the movement of data, (4) using a high-speed CMOS with one micron technology, and (5) having the whole MCAP on a single MCM-D.

2 Modularly Configured Attached Processor (MCAP) Architechture

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components [5], [6]. This standard set consists of three types of asynchronous connections and twelve types of components. These component types are such that each member of the class may include parallel processing, memory to memory pipelines, and be constructed in a building block fashion. They encompass routing as well as memory, control and processing components. By overlapping processing with memory accessing and matching an architecture with a set of algorithms, the average sustainable rate for a specific set of algorithms can attain at least 60% of the peak rate. These rules allow the components to be easily configured in different ways, thereby allowing the construction of attached processors that efficiently perform different sets of algorithms.

An example architecture is given in Figure 1. It's processing subsection includes a comparator, a negator (elementary component), a reciprocator (elementary component), a set of four pipelined adders capable of accumulation (via feedback), and a set of four pipelined multipliers. Each adder and multiplier is

constructed of four stages (a two-input component followed by three elementary components). All communications to and from the processing components are through six link components, three on each side of the processor. Join and fork components are provided to allow flexible use of the link components. There is a dual access component to provide intermediate memory and a connection to main memory. The single access component provides internal storage.

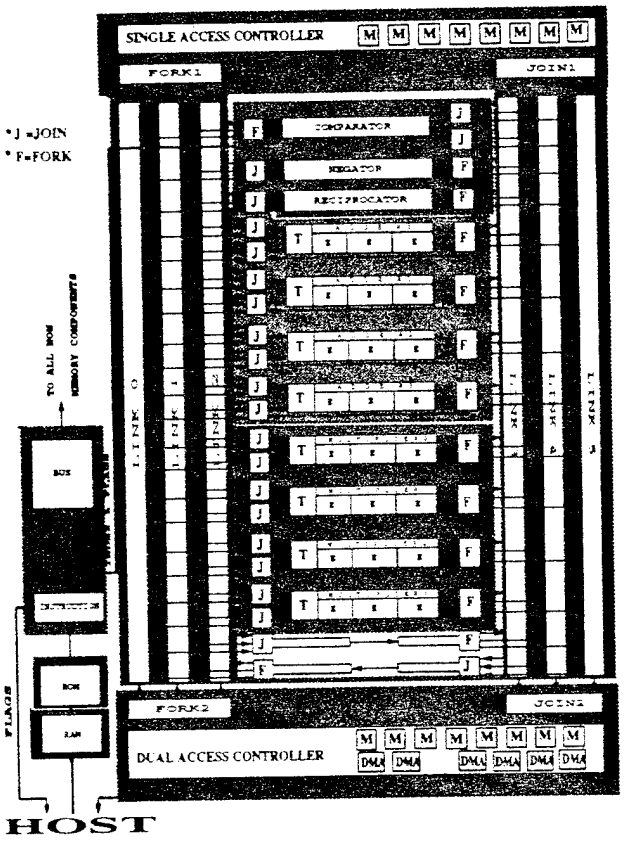


FIGURE 1. AN EXAMPLE MCAP ARCHITECHTURE

3 Design Considerations and Results

In our evaluation, CMOS was picked as the benchmark logic technology because of its commercial maturity. In the future, we plan to evaluate other faster technologies such as GaAs, BiCMOS, and ECL. Since the signal delays associated with the conventional printed circuit board (PCB) implementation are expected to be prohibitively excessive, it was decided that the fabrication of an MCAP on a Multi Chip Module (MCM) or Wafer Scale Integration (WSI) are the only realistic alternatives for attaining high performance. WSI integrates an undiced wafer of defect tolerant VLSI chips with global power, clock, and signal distribution networks. MCM technology on the other hand has discrete VLSI chips, probably of different types, mounted on a substrate that supports global power, clock and signal distribution networks. Since the chips are procured separately, the substrate can be optimized and tested independantly before assembly. Therefore defect tolerance is not required. However, the "known good die" problem is yet to be solved. Further, multichip modules are classified according to the substrate technology: MCM-Ceramic, MCM-Deposition, and MCM-Laminated. In this work we used MCM-D for design evaluation. MCM-D manufacturing processes are similar to those used in the semiconductor industry and can be used to achieve high densities and fine line geometries. In the physical design, we must face the traditional problems in placement and routing required by the high performance systems. As the clock frequency is increased, we need to account for transmission line effects due to long interconnections. Parasitics on the interconnects, inductances on the power lines, and the I/O pin limitation are the three vital shortcomings of current packaging technologies, which could be tackled by (a) minimum chip to chip interconnections, (b) high interconnection density, and (c) parallel architecture. Other factors which need to be considered are ground and power plane generation and physical design verification. The thermal considerations are a direct result of the substrate type, bonding selection, and the placement of chip dice.

A system level model, referred to as the SUSPENS model (Stanford University System PErformaNce Simulator) [7] is used to predict the performance of the MCAP. This model emphasizes the interactions among devices, circuits, logic, packaging, and architecture. The same model could be used in future to compare logic technologies (e.g. CMOS, Bipolar, and GaAs) and various packaging technologies (e.g. MCM-D, MCM-C, PCB, and WSI).

3.1 Chip Level Design

3.1.1 Transistor Count

To illustrate the method used to estimate the total number of transistors in the example MCAP (Fig. 1), we will consider one of the floating point adders. Each adder has four pipelined stages and uses the IEEE double precision standard. Further this adder could be broken down into: (a) nine 64-bit registers with 4032 transistors, (b) seventy-four 2-input XOR gates with 592 transistors, (c) one hundred and twenty-six 2-

to-1 MUXs with 504 transistors, (d) two 11-bit adders with 528 transistors, (e) one 52-bit adder with 1248 transistors, (f) a 64-bit leading zero detector with 5000 transistors, (g) two 52-bit barrel shifters with 4000 transistors, and (h) rounding and other control logic taking 6500 transistors.

ELEMENT	DESCRIPTION	# TRANSISTORS
Memory Element	Has 4 k each of RAM and ROM	1.84 M
Instruction	Has 8 words of FIFO	10.0 K
Bus	Has 8 words of FIFO	10.0 K
Elementary	Has 8 words of FIFO	10.0 K
Two input	Has 8 words of FIFO	12.0 K
Join	3 inputs and 8 FIFO	14. 0 K
Fork	3 inputs and 8 FIFO	7.0 K
Link	4 inputs and 5 outputs	25.0 K
Static Ram	16 elements of 1k each	6.30 M
Single Access Controller	Controls 8 Memory elements	11.0 K
Dual Access Controller	Controls 8 memory elements and 6 DMA channels	61.0 K
Compare	sends out Flags and Indices	25.0 K
Reciprocate	using Convergence method	50.0 K
Negate	invert the sign bit	1.0 K
Fl. Pt. Adder	using CLA's, Barrel shifters	23.0 K
Fl. Pt. Multiplier	using Modified Booth's Alg.	61.0 K
MCAP	Total # of Transistors	9.85 Million
MCM	With 25 Chips and 600 I/O 's	9.85 Million

TABLE 1. TRANSISTOR COUNT FOR THE VARIOUS MCAP COMPONENTS.

Thus the total number of transistors for the above adder is approximately 23K. Similarly, the transistor count for the pipelined 64-bit floating point multiplier is approximately 61K (based on a modified Booth's algorithm). Likewise, the transistor count for the other elements in the MCAP were calculated and the results are presented in Table 1. The resulting number of transistors for the whole MCAP is approximately ten million.

3.1.2 Output Driver Design

In the proposed MCAP architecture, the bottle neck is the communication through link, single-access, and dual-access components because of their high fanout and large interconnection lengths. This means that the output buffers for these elements must be relatively large. We present the delay, area, and power dissipation calculations for the buffers as functions of fanout (F) and interconnection length (ℓ).

For the chip level model, we have assumed that the input capacitance of a gate (including the lead and ESD capacitances) is C_{in} = 1 pF. Additional parameters [7] are in Table 2.

Parameter	MCM-D
Pw (uM)	50
Nw	2
Wint (uM)	25
Hint (uM)	2
@ 1 Ghz (uM)	2
Rint (ohm/cm)	3.4
Dielectric constant	3.4
Vm (cm/nsec)	16
Cint (pF/cm)	1.0
Z0 (ohm)	60
Cpad (pF)	0.25
Pp (uM)	100

TABLE 2. THIN FILM HYBRID PARAMETERS.

Average Delay

In general, the minimum size of a logic gate has a W/L ratio of 2. Therefore, we began with a ratio of 2 and, by stages, moved to higher values in order to drive a load in a small amount of time.

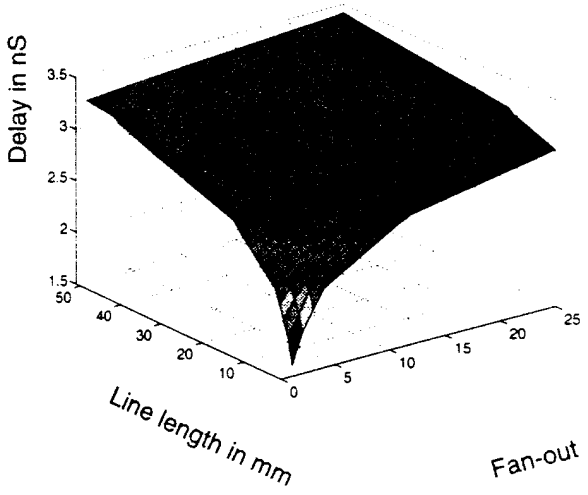


FIGURE 2. BUFFER AND INTERCONNECT DELAY.

By dividing the driver into a number of buffers with increasing W/L ratio, optimum speeds can be achieved. It has been found that a stage ratio of e

[8] gives best results. We have used a stage ratio of 3 for simplicity. The optimum number of stages (N) is dependant on the load capacitance (C_l). The relationship is

$$N = 0.91(\ln C_l + 4.19)$$

where N is truncated (rounded down) to the nearest integer.

Using the optimum number of stages, the average delay is

$$T_{avg} = 0.484(N-1) + 5C_l/3(N-1) + 0.076 \text{ ns}$$

Delay calculations [9] are shown in Figure 2.

Buffer Area

A simple inverter with $(W/L)_n = (W/L)_p = 2$ will need an area of 171 λ^2. A buffer with equal rise (t_r) and fall (t_f) times requires $(W/L)_p = 2(W/L)_n = 4$ and the area is going to be 203 λ^2. The total area of the buffer depends on the number of stages and, hence, is a function of F and ℓ. We have

$$Area \approx 220 \times 3^{N-1} \ \lambda^2$$

Area calculations [9] are plotted in Figure 3.

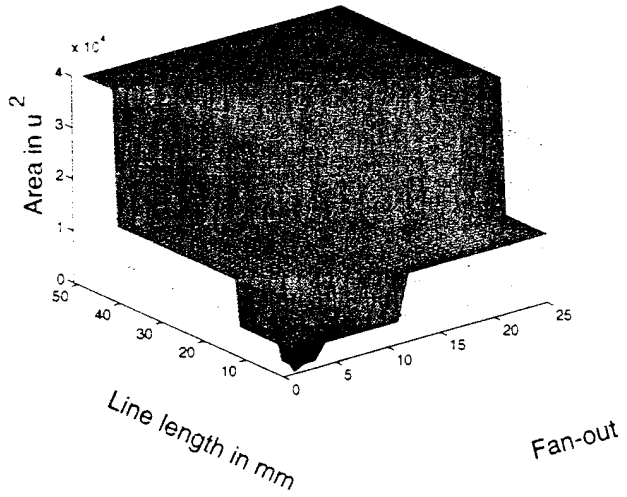


FIGURE 3. AREA OF THE BUFFERING STAGE.

Power Dissipation In The Buffer

In CMOS, most of the power is dissipated during switching and, hence, dynamic power is approximately equal to the total power. The dynamic power is

$$P_d = C_T \times v^2 \times f_{avg} = v^2(C_l + C_{buff})/T_{avg}$$

$$where \ C_{buff} = 0.0152(3^{N-1}) \ pF.$$

Power dissipation calculations [9] are presented in Figure 4. Since the design of an MCAP uses asyn-

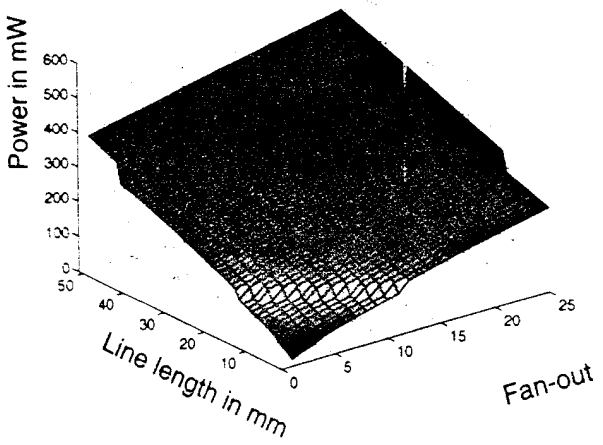


FIGURE 4. POWER DISSIPATED IN THE BUFFER.

chronous communication, the transfers over a link component involve the return of an acknowledge signal and the transmission of an output enable signal. It is estimated that the transfer rate may be as high as $f = 1/2[T_{avg} + T_{chip}]$ Hz (where T_{chip} is the delay of the chip and T_{avg} is the delay on the interconnection)

Load Capacitance

For the load capacitance

$$C_l = C_{int_{tot}} + F \times C_{in},$$

with

$$C_{int_{tot}} = C_{int} \times \ell = \ell \text{ pF}$$

where ℓ is in cm and $C_{int} = 1$ pF/cm. Therefore,

$$C_l = (\ell + F) \text{ pF}.$$

The resistance of the interconnect is

$$R_{int_{tot}} = R_{int} \times \ell \ \Omega \ .$$

3.1.3 Modified SUSPENS Model

Given (a) the approximate number of logic gates, (b) the transistor technology parameters, (c) packaging technology parameters, and (d) the number of pads per chip (estimated from Rent's rule), the SUSPENS model can estimate system performance. Rent's rule is an empirical result obtained by observing existing

designs. The design philosophy and methodology affect Rent's constants. If the predictions are made for a system with an entirely different design philosophy from the one from which Rent's data were obtained, the results will have little meaning. The SUSPENS model, as originally proposed, used Rent's constants, therefore we have developed constants that are applicable to the novel architecture of the MCAP. Our

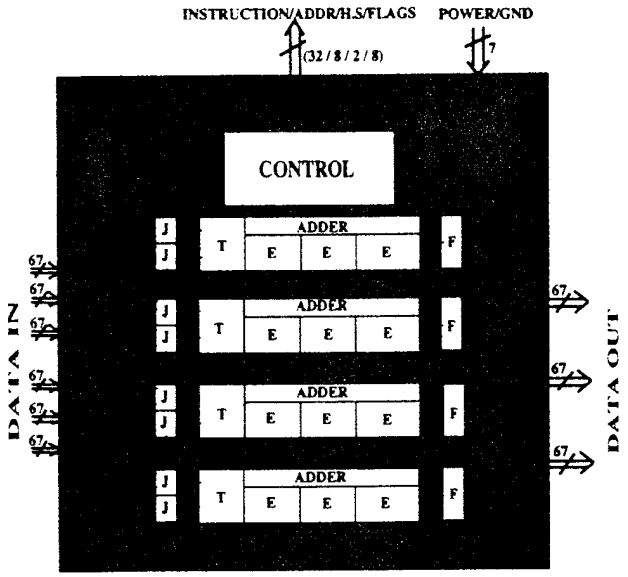


FIGURE 5. BLOCK DIAGRAM OF THE ADDER CHIP

approach to developing these constants was to determine the area needed for an inverter (with equal rise and fall times) and a carry generator circuit. From this we computed the average area per transistor. Thirty percent of that area was assumed to be taken by the interconnections, resulting in a figure of 203 λ^2 per transistor. This result was used as input to the SUSPENS model for computations done at the chip level. The I/O buffer areas were estimated separately as presented in section 3.1.2. To illustrate the use of the SUSPENS model using table 3 [7], we will consider the adder chip with a 1 μ technology (see Figure 5). The adder chip contains four 64-bit floating point adders.

The input stage delay is

$$T_i = f_g \cdot \frac{R_{tr}}{K_i} \cdot 3K_oC_{tr}$$

f_g (number of n – transistors in series) = 3

K_i (W/L ratio of input transistors) = 4

K_o (W/L ratio of output buffer) = 4

C_{tr} (for 1 μm CMOS) = 3 fF

R_{tr} (for 1 μm CMOS) = 15 K

T_i = 0.41 ns

The output stage delay is

$$T_o = \frac{f_g R_{tr}}{K_o} \cdot [\ell_{av}C_{int} + K_iC_{tr}] + \ell_{av}R_{int}[\frac{\ell_{av}C_{int}}{2} + K_iC_{tr}]$$

ℓ_{av} (average interconnection length)

C_{int} = 2 pF/cm

R_{int} = 375 Ω/cm

The total gate delay, $T_g = T_i + T_o$, is 0.84 ns. The delay for an adder is

$$T_{chip} = f_{ld}T_g + R_{int}C_{int}(D_c^2/2) + (D_c/v_c)$$

restrict the logic depth, f_{ld}, to 6

$v_c = 2.5 \times 10^{12}$ cm/sec

D_c = 0.3 cm

$$T_{chip} = 6(0.84 \times 10^{-9}) + 375(2 \times 10^{-12})(0.3^2/2) + (0.3/2.5 \times 10^{12})$$

= 5.07 ns (which includes the latch time, logic time, setup time, and clock skew)

Thus the maximum frequency of the adder chip is 197 MHz. By incorporating pipelining and recalling that the total chip area actually has four of these floating point 64-bit adders, the throughput is improved by a factor of more than four.

Parameter	CMOS
Leff	1.0
tgox (A)	250
Vdd (V)	3.3
Rtr (ohm)	15,000
Ctr (fF)	3.0
Wint (uM)	2.0
Wsp (uM)	2.0
Hint (uM)	0.4
pw (uM)	4.0
nw	3
Rint (ohm/cm)	375
Cint (pF/cm)	2.0

TABLE 3. 1 micron TECHNOLOGY PARAMETERS

The external capacitance of a gate is

$$C_{ext} = f_g\ell_{avg}C_{int} + f_gK_iC_{tr} = 36.5 \text{ fF},$$

where $\ell_{avg} = 81.3 \times 10^{-6}$ cm. The internal capacitance of a gate is

$$C_{int} = 3K_o C_{tr} + 5C_{tr} = 51 \text{ fF}$$

The total capacitance per logic gate, $C_g = C_{ext} + C_{int}$, is 87.5 fF.

3.2 Interconnection and Packaging Considerations

Once the results for each technology have been obtained (see Table 4), the package level model is incorporated using MCMs and WSI (Table 2).Layout of MCAP in a MCM configuration is shown in Fig.6. The average interconnection length at the module level (in units of chip footprint size) is

$$\begin{aligned} \bar{R}_m &= \frac{2}{9}[7\frac{N_c^{\eta-0.5}-1}{4^{\eta-0.5}-1} - \frac{1-N_c^{\eta-0.75}}{1-4^{\eta-0.75}}]\frac{1-4^{\eta-1}}{1-N_c^{\eta-1}} \\ & N_c = 4 \\ & \eta = 0.65 \ (see[7]) \\ \bar{R}_m &= 1.33 \end{aligned}$$

Component	# of Tran's	Area (cm^2)	# of i/o's	fc (Mhz)	Pc (watt)
Adder	176 k	0.41	660	131	2.75
Multiplier	328 k	0.41	660	122	3.44
C-N-R	118 k	0.55	874	139	2.54
SRAM (4k)	1.57 M	0.58	150	110	8.66
SRAM (16*1k)	393 k (* 16)	0.16	150 (*16)	110	2.36
ROM	262 k	0.11	80	110	1.66
D-control	82 k	0.56	846	120	3.67
S-control	32 k	0.14	222	114	0.70
Instn/Bus	20 k	0.14	200	135	1.65
Link	15 k	0.14	200	123	1.27
Module	10 M	42.0	600	100	68.82

TABLE 4. PARAMETERS OF THE MCAP ON MCM-D.

(1 micron PROCESS)

As explained in section 3.1.2, the output driver is designed for a critical length of 1.5 cm and fanout of 3. From Fig. 2, the buffer delay is found to be 2.45ns. Further, the capacitive delay (caused by the loading of the I/O pins and contact pads) and the time of flight delay on the transmission lines are both calculated (for a 1 μ technology) using

$$T_{additional} = 2 \cdot Z_o \cdot C_{pad} + L_{int}/v_m = 0.124 \text{ ns}$$

Adding in $T_{chip} = 5.07$ ns, we determine that the total delay of the adder chip = 5.07 ns + 2.45 ns + 0.124 ns = 7.64 ns. Thus, the adder chip can output at the rate of 131 MHz.The dynamic power dissipation per gate is based on the maximum adder chip operating frequency (f_c) and the percentage of gates that switch during a clock period (f_d). The dynamic power dissipation is

$$\begin{aligned} P_g &= \frac{1}{2} f_c f_d C_g V_{DD}^2 \\ &= \frac{1}{2}(131 \times 10^6)(0.3)(83.5 \times 10^{-12})(3.3)^2 \\ &= 56.3 \ \mu\text{W} \end{aligned}$$

The power dissipation of the chip is the product of the number of gates (N_g) and the dynamic power dissipation per gate (P_g).

$$\begin{aligned} P_c &= N_g \cdot P_g \\ &= (176k/4) \cdot (56.3 \times 10^{-6}) \\ &= 1.65\text{W} \end{aligned}$$

Thus the power density for the adder chip (area = 0.09 cm^2) is 18.4 W/cm^2. The parameters for other chips are calculated as described above, and Table 4 gives the results. A similar procedure is repeated for 0.5 μ and 0.25 μ technologies. The area required by the output buffers are added to the transistor area to get the die area. Assuming an area distributed solder bumps with 100 μm diameter and 250 μm pitch. The footprint of the Adder chip die is found to be 0.64 cm.

The module frequency is, therefore appoximately 100 MHz (considering the processing and driving involved in one cycle). The module size is 9.0 cm X 10.0 cm. Module power dissipation (P_m) is determined by

$$C_m = \frac{F_c}{1+F_c} N_c N_p (3\frac{1-5^N}{1-5} C_{tr} + 2C_{pad} + \bar{R}_m F_p C_{int})$$

$$C_m = 63.4 \ nF$$

$$P_m = \frac{1}{2}(F_D)(f_s)(C_m)(V_{DD}^2) = 20.7 \ W$$

The actual power dissipation is the greater of P_m and the sum of the power dissipated at all the chip dice. Thus we calculate the power dissipation for the module to be 68.82 W.

4 Conclusions

Design evaluations for implementing a novel modularly configured attached processor architecture using CMOS logic on an MCM-D (Fig 6.) revealed that approximately ten million transistors will be needed.These could be placed on a set of twenty five chip dice. Delay, area, and power calculations were done with the SUSPENS model (however, Rent's rule was not used). Delay calculations(including logic

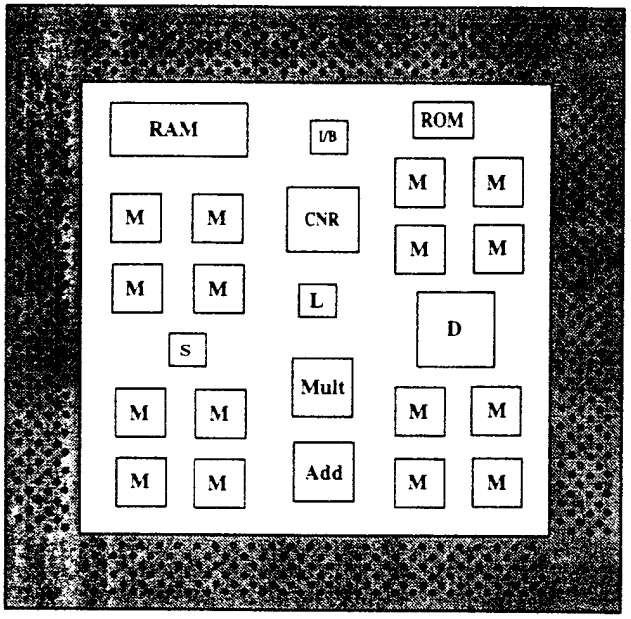


FIGURE 6. LAYOUT OF MCAP ON MCM

(All the die areas have been optimized for 15 W/sq.cm)

# of Transistors = 10 Million

# of I/Os = 600

delay, interconnect delay and output driver delay) showed that the MCAP module, on average, would achieve speeds in the 150 MFLOPS range. The single-access memory controller component chip (S-control) was found to be the slowest, 110 MHz for 1 μm, 160 MHz for 0.5 μm, and 220 MHz for 0.25 μm CMOS. The areas were optimized for power densities, low enough to allow air cooling. Higher speeds would be achievable with faster logic like BiCMOS, ECL, and GaAs. Power dissipation calculation showed that approximately 70 watts will be dissipated in the MCAP module and air cooling would suffice for the 1 μ design rule.We are in the process of performing further design calculations involving Wafer Scale Integration(WSI)and GaAs technology.

5 Acknowledgements

The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency. Thanks are due to Swaroop N. Kumar for his help in the revision of this manuscript.

References

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1985.

[2] J. A. Swanson, G. R. Cameron, and J. C. Haberland, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *IEEE Computer*, vol. 16, no. 6, pp. 85–91, June 1983.

[3] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

[4] J. H. Tang and E. S. Davidson, "An Evaluation of Cray I and Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," *Proc. 1984 Int'l Conf. on Supercomputing*, pp. 510–518, 1988.

[5] G. A. Gibson, "Investigation of Modularly Configured Attached Processors with Intelligent Memories," *Technical Report to Office of Naval Research*, (Grant No. N00014-93-1-1343), March 31, 1994.

[6] G. A. Gibson, V. P. Singh *et al* "Application and implementation of a modularly configurable attached processor," *To be presented at the International Symposium on High-Performance Computer Architecture*, Taiwan, December, 1994.

[7] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, 1990.

[8] K. E. N. Weste, *Principles of CMOS VLSI design - A Systems perspective*, Addison Wesley, 1993.

[9] S. J. Singh, "A comparative evaluation of implementing a novel MCAP architechture," *M.S. Thesis*, E.C.E Department, The University of Texas at El Paso, December, 1994.

ATTACHMENT D

SIMULATION AND FAST PROTOTYPING OF MODULARLY CONFIGURABLE ATTACHED PROCESSORS

G. A. Gibson, A. Brito, Y. C. Chang, D. Saenz, and E. Castro

Department of Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, Texas 79968-0523

Abstract

A broad class of attached processors (MCAPs) that are constructed from a standard set of connections and components is defined and a simulation package (SIMARC) for evaluating members of the class relative to specified sets of algorithms is described. Together the MCAP connection and component definitions and SIMARC package provide a fast prototyping means when designing efficient attached processors for executing computationally intense algorithms. A dynamic multicolored graphical display during a simulation facilitates the detection of bottlenecks and a graphical editor permits easy modification of an architecture. Use of SIMARC requires specification of the architecture and writing assembler-level programs for executing the algorithms of interest. An example is presented in which SIMARC is used to design an MCAP that achieves 95% efficiency (for the processing logic) while performing matrix multiplication. [1]

1 INTRODUCTION

Modern design procedures include analytical modeling, simulation modeling, simulation evaluation and prototyping [1], with analytical modeling being the mathematical and statistical analysis of a relatively crude system model. This phase of the design process is the pencil and paper approach that can be done quickly, but is the least accurate in predicting the performance of the final product. Simulation modeling requires a much more detailed level of system specification, a level that is sufficient to provide the design input to the computer simulation program, or simulator, that is to be used. Test data is then selected and the simulator is used to produce a more accurate estimate of performance than can be obtained from analytical modeling. The accuracy of this estimate depends on the level of detail included in the simulator, the quantity of test data and the care used in choosing the test data. Simulation modeling and evaluation are iterated until satisfactory results are obtained, at which time a prototype is built. If the prototype needs only minor adjustments, the design team may proceed with the production design; otherwise, it must return to an earlier phase and repeat the design process. Because prototyping is normally expensive, it is important that the simulations be extensive enough to avoid such returns. However, it is also important that the prototyping stage be reached as quickly as possible, and much attention has recently been given to developing simulation and performance evaluation systems that permit fast prototyping.

For computer system design, simulation may be performed at several levels, ranging from the major component level, to the module level [printed circuit board, multichip module (MCM) or wafer, whichever is applicable], to the register transfer level, to the gate level, and then to the transistor level. As one proceeds from the major component level to transistor level, clearly the amount of detail increases and the time needed to execute a simulation increases accordingly. Therefore, there is a tradeoff between simulation time and accuracy. Normally, top-down design would be used, which employs a component level simulator to design the overall system and then progresses downward to the transistor level, at which the layouts of the individual integrated circuits are done independently.

To reduce the complexity of a simulator and quantity of specifications required by it, simulators at the major component and module levels are restricted in various ways. They are normally restricted to a class of architectures and sometimes to a particular type of computer, which serves to specify the instruction set. Eleven such performance evaluation tools that relate algorithms and architectures and consider testing of programs to meet real-time constraints are summarized in [2]. Others are described in special issues of the *International Journal of Computer Simulation* [3], [4].

As always, the accuracy of the timing information gathered depends on the level of detail being simulated. Some simulators are very specific in their application. The advantage in restricting the use of a simulator is, of course, that highly reliable performance data can be gathered in a reasonable amount of simulation time. Of interest here is an architecture

[1] The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

simulation package, SIMARC, that simulates a fairly broad class of attached processors referred as Modularly Configurable Attached Processors (MCAPs). The primary advantage of an MCAP is that it is constructed from a standard set of connections and components that can be easily fit together to match the set of algorithms it is to execute. The SIMARC package allows the quick reconfiguration and simulation of MCAP architectures so that an MCAP design with high utilization of its logic for a given set of algorithms can be achieved within a short time. Because MCAPs are connected to host processors, there is no need to simulate operating system functions. Also, because MCAPs use memory-to-memory pipelines, they are inefficient when performing algorithms that include a considerable amount of decision-making (e.g., a binary search). Therefore, emphasis is placed on computationally intense algorithms such as those for performing matrix operations, signal processing and image processing and for solving simultaneous linear equations, partial differential equations and ordinary differential equations.

Section 2 gives a brief MCAP definition and the next two sections describe the simulator and provide an example. The last section serves as a summary and indicates future improvements to the simulator.

2 MCAP DEFINITION

An attached, or back-end, processor is a processing system that is connected to a host computer for the purpose of very quickly executing most of the overall system's computational tasks. Typical early attached processors were the AP-120B and FPS-164 made by Floating Point Systems, Inc., the IBM 3838, and the MATP made by Datawest, Inc. [9]. Although the early attached processors included limited multiprocessing, the more recently implemented processing arrays are also controlled by a host (e.g., the PAX computer [10]) and are designed to perform most of the overall system's computational tasks. The specific purpose of an attached processor is to execute members of a set of algorithms very quickly. The broader the set of algorithms the more generally applicable the attached processor. The underlying goal of the designer is to efficiently utilize the hardware for as broad a set of algorithms as possible. By using the MCAP building block approach along with the SIMARC package, efficient matches between architectures and sets of algorithms are easily established.

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components. This standard set consists of two types of asynchronous connections and twelve types of components. The definitions of the connection and component types provide a standard set of rules that allow the components to be easily configured in different ways to construct attached processors that can efficiently perform different sets of algorithms.

An MCAP has exactly one instruction component and it is connected to a memory component for storing instructions. Most of this memory component is a ROM that contains the subprograms needed to execute the algorithms, but some of it is a RAM that can receive instructions (those that initiate the subprograms) from the host.

An MCAP operates by drawing an instruction stream from the instruction memory component into the instruction component. The instruction component uses internal instructions in the stream to form external instructions that are then distributed to the other non-memory components through the MCAP's bus component. The external instructions are for setting up and supervising the interconnected memory-to-memory pipelines within the MCAP that perform the operation needed to complete an algorithm (or phase within an algorithm). The instruction stream is illustrated in Fig. 1. Note that all components in the instruction stream include input instruction queues. When the non-memory components have received all of the instructions needed to perform an algorithm, they automatically prefetch the data from the memory components, route the data to and from the processor components and store the results back into the memory components. All non-memory components have input data queues. Some controller components, which are the components that supervise all memory accessing, are used to automatically transfer data between the host's main memory and the MCAP's memory components. The instruction and data streams are separate, thereby allowing the instructions needed for the next algorithm to be distributed while the current algorithm is executing.

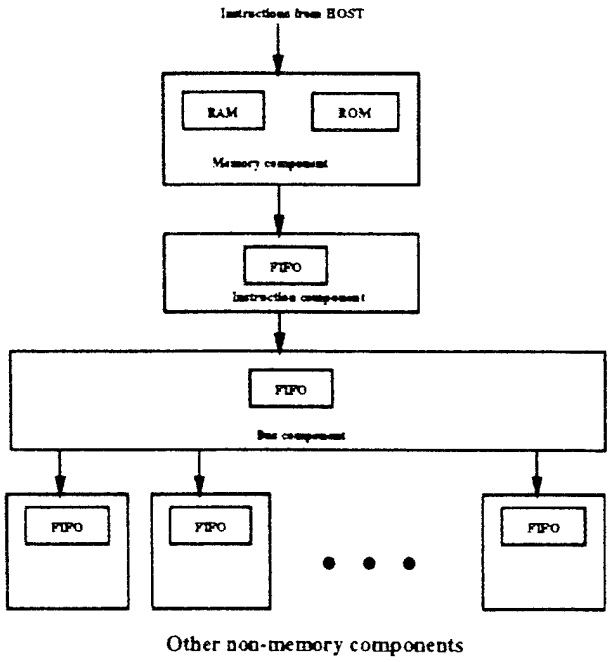


Fig. 1. The instruction stream

The two types of connections are referred to as instruction and data connections. All connections are unidirectional and asynchronous. Memory components are considered

to be integral parts of controller/memory subsystems and the design of the connections between the memory components and their controllers is left to the designer of these subsystems.

Instruction connections are for passing instructions from the instruction component to the bus component and from the bus component to one of the other non-memory components. An instruction connection consists of unidirectional instruction and address buses and a Req/Ack handshaking pair. The component that is to receive the instruction is indicated by the a component number on the address bus. Data connections are used to pass data between components and consist of only a unidirectional data bus and a Req/Ack pair. All transfers include the latching of an instruction or datum into a queue at the receiving end.

The twelve types of components are divided into six categories as indicated below:

- Instruction (I)
- Bus (B)
- Memory (M)
- Processor
  - Elementary–one input, one output (E)
  - Two-input–two inputs, one output (T)
  - Comparator–two inputs, one output plus special outputs (C)
- Router
  - Join–multiple inputs, one output (J)
  - Fork–one input, multiple outputs (F)
  - Link–multiple inputs, multiple outputs (L)
- Controller
  - RAM–internal to MCAP, no partitions (R)
  - Single-access–internal to MCAP, has partitions (S)
  - Dual-access–connects to main memory, has partitions (D)

The letter used to indicate each type of component is given in parentheses.

As mentioned earlier, an MCAP contains one memory component for storing instructions, one instruction component for executing internal instructions and forming external instructions, and one bus component for distributing the instructions. An MCAP may contain several controller, router, and processor components and several other memory components for storing data. However, the other memory components can be connected to controller components only. Only controller components are capable of being programmed to prefetch data from and deposit data into data memory components.

Each non-memory component that is used during the execution of an algorithm contains an instruction input queue, one or more data input queues, and control logic that includes a number of registers. The instructions for an algorithm received by a component fill these registers and then the register contents dictate the activity within the component while the algorithm is executed. They determine the component's mode and, for a routing component, the patterns for accepting inputs and distributing outputs. For a controller component, they determine the memory partitions and patterns for prefetching operands and storing results.

Each of the components that receives instructions contains a Number of Operands Output (NOO) register that is always the last register filled before the component begins its part in the execution of the algorithm. Each time the component outputs an operand, the NOO register is decremented. When the NOO register becomes zero, the component has completed its part in executing the current algorithm (or phase). It may then distribute new values, those needed for the next algorithm, from its instruction input queue to its registers. This cycle may continue indefinitely. Except for reacting to the handshaking (i.e., Req and Ack) signals in its connections, each component acts independently. The data is input to a data queue through an input connection, processed or routed through a bus, and output through an output connection.

All controller components have an output data connection for outputting operands to the remainder of the MCAP and an input data connection for inputting results from the MCAP. Therefore, they must be capable of handling both an output data stream and an input data stream. A queue is inserted in each of these data streams. In addition to the NOO register, a Number of Operands Input register is needed for the input stream. A D controller also has a second set of input and output connections.

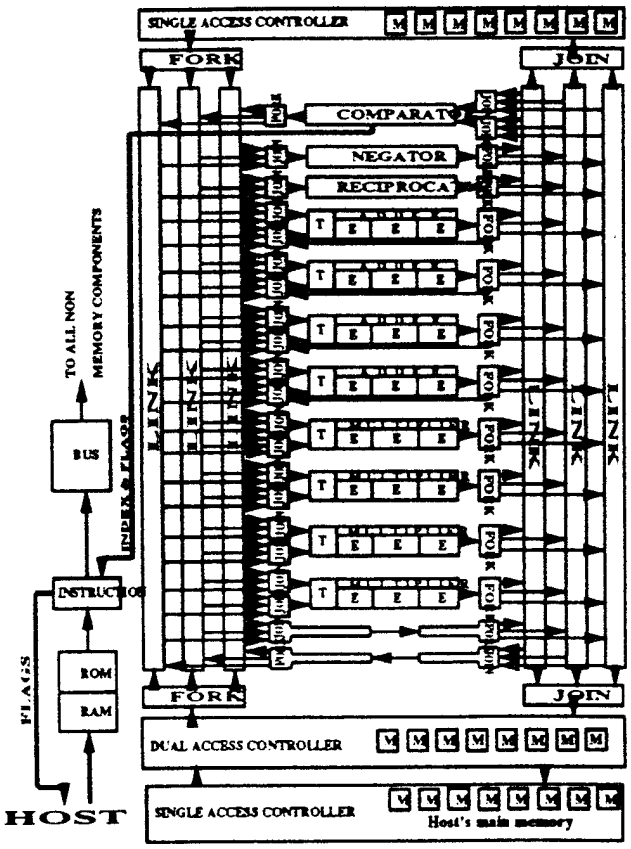


Fig. 2. An Example MCAP Architecture

For a more complete description of an MCAP refer to [11]. An example architecture is given in Fig. 2. Its processing subsection includes a comparator (C component), a negator (E component), a reciprocator (E component), a set of four pipelined adders capable of accumulation, and a set of four pipelined multipliers. Each adder or multiplier is constructed of four stages (a T component followed by three E components). All communications to and from the processing components are through six link (L) components, three on each side of the processor. Join (J) and fork (F) components are provided to allow flexible use of the L components. Also, to allow for accumulation there is a feedback connection between the F component at the output from each adder and the J component at the input to the adder. There is a dual-access controller (D) component to provide intermediate memory and a connection to the single-access controller (S) that controls main memory. A second S component provides additional internal storage.

3 SIMARC PACKAGE

SIMARC is a menu-driven package written using C++ and is for PC compatible computers operating under DOS. By using C++ both the components in an architecture and the instructions in a program could be defined and operated on as objects.

The programs in the SIMARC system communicate through a collection of files. The relationships between the programs and the files are summarized in Fig. 3. The files are:

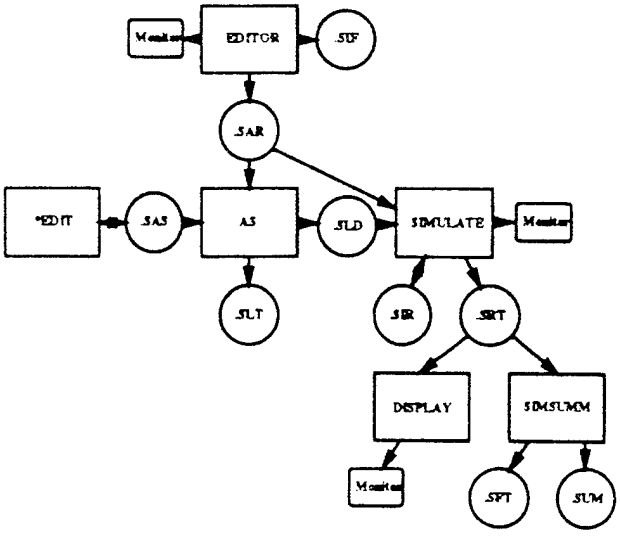
Architecture (.SAR) -	a complete description of an architecture
Information (.SIF) -	a textual description of the architecture.
Program (.SAS) -	a source program in the form needed by the assembler.
List (.SLT) -	a list of the errors produced during an assembly.
Load (.SLD) -	a program in the form needed by the simulator.
Result (.SRT) -	the results produced by a simulation.
Irregular intervals (.SIR) -	specifies the time at which results are to be taken.
Fractional result (.SFT) -	same as an .SRT file except results are percentages.
Summary (.SUM) -	summary information from succesive tests.

The programs are:

EDITOR -	creates a new architecture or changes an existing architecture.
AS -	assembles source program into load form that can be used by SIMULATE.
SIMULATE -	simulates a program on an architecture and accumulates results.
DISPLAY -	displays the results of a simulation on the monitor.
SIMSUMM -	produces summary information and stores results as percentages

When using the SIMARC package, the design procedure is to:

1. Determine the set of algorithms around which the MCAP is to be optimized, the technology and packaging to be used and the MCAP's initial architecture.
2. Use EDITOR to graphical enter the architecture's description.
3. Use EDIT to write, or rewrite, programs for the algorithms.
4. Assemble the programs using AS and the architecture file created by Step 2.
5. Simulate the programs using SIMULATE and determine the bottlenecks by examining the graphical output and the result files.
6. Redesign the architecture to eliminate the bottlenecks as much as possible within the constraints of the technology and packaging.
7. Repeat Steps 2 through 6 until the design is optimized.



* Although the SIMARC package uses the DOS program EDIT, any text editor capable of producing an ASCII file, could be used to produce a .SAS file.

Fig. 3. Relationship between the SIMARC package's programs and files.

From other simulation packages [12,13], the value of graphical output during simulation and graphical specification of a system so that it can be easily created and modified is well established. It is seen from Fig. 3 that *EDITOR*, *SIMULATE* and *DISPLAY* display information on the screen. The display is of the architecture specified while executing *EDITOR*. In the display each component is shown as a collection of rectangles which may contain up to four characters of text. Normally, the text is an identifying mnemonic that has been assigned to the component at the time it is created, but it may indicate the component's type, percentage of time in a particular state, or the component's identifying number. A component may also be displayed using a color that represents its type or current state. In addition, *EDITOR* can show an architecture's connections.

Upon starting the execution of SIMARC, the main menu appears. To create or edit either an architecture or program the user would respond by typing "E", which would cause the edit menu to appear. This menu would allow the user to create or edit an architecture or program, assemble a program, or view the errors resulting from an assembly.

3.1 EDITOR Program

The program *EDITOR* is for creating or modifying an architecture file and is initiated by typing "E" in response to the edit menu. It first requests the name of the architecture file. If the file does not exist, then the architecture is to be created; otherwise, an existing architecture is to be modified.

An architecture is defined by the:(1) attributes of its components, and (2). connections between its components. There are two kinds of attributes. There are graphical attributes for displaying the component and physical attributes that give the component's characteristics (its execution time, queue lengths and so on). When an architecture is created, a menu of icons that appears on the right side of the screen is used to enter the component attributes and connections. Because all MCAPs require exactly one each of the instruction. bus and instruction memory components, and the connections between these components and the other components must always be the same, the user must enter only the attributes of these components. Their connections are made automatically. For all other components, the user would typically enter the physical attributes by responding to a series of prompts, shape the component by defining its rectangles, move the component to the position it is to be displayed and make the desired data connections between it and the other components. A component's graphical attributes are determined during the shaping and moving actions. EDITOR includes an easy means of duplicating a component or flipping a component's image with respect to either the horizontal or vertical axis. Modifying an architecture may consist of simply changing the attributes of some of the components, deleting or adding components, or deleting or adding connections. If a component is deleted, all connections to the component are automatically deleted.

3.2 EDIT and AS Programs

The DOS program EDIT is used for creating and editing source program files. It is initiated by typing "E" in response to the edit menu. The language used to create a source program is at the assembler level with labels used to specify instruction addresses and mnemonics used to specify instruction types. In addition, the mnemonics assigned to the components when the architecture is created are used to specify the components that are to receive the instructions, sources of the input connections, and destinations of the output connections.

The program AS is initiated by choosing "A" from the edit menu. The user then enters the names of the architecture and source program files. As the assembly takes place any errors that occur are recorded in an error file and, at the end of the assembly, the total number of errors is displayed. By pressing <ENTER> a return is made to the edit menu. By choosing "V" from the edit menu, the program EDIT may be used to view the contents of an error file.

3.3 SIMULATE Program

A flowchart of the program *SIMULATE* is given in Fig. 4. When "S" is typed in response to the main menu, the simulation begins by querying the user for the names of the architecture, program and result files. Also, it asks the user whether the results are to be recorded at regular or irregular intervals and, if regular intervals are chosen, the length of the interval is requested. If irregular intervals are chosen, the user must either enter the times at which results are to be recorded or the name of an irregular interval file. *SIMULATE* then enters its main loop.

SIMULATE assumes that the possible states that a component can be in are as defined below. The component types that can take on a state are given in parentheses.

BUSY (all component types) - component is actively performing a function.

WAIT (all component types) - component is waiting for its output to be taken or, for the I component, a flag to be set.

IDLE (I,E,T,C,J,F,L,R,S,D) - component is waiting for input.

FREE (all component types) - component is completely inactive.

DIST (E,T,C,J,F,L,R,S,D) - component is distributing instructions to registers.

Each time the main loop is executed the states of all components are updated according to their current states, including the states of their queues, and the Ack inputs from their output connections. The state diagram for components that can take on five states is given in Fig. 5. Also, all queues are updated according to their current states, Req inputs from their input connections and whether or not they have been popped by the components containing them. No more

than one state change can occur in a single execution of the main loop. Each time around the loop to consume one basic increment of time and system time is measured in terms of the number of basic increments (i.e., loop executions) that have occurred since the beginning of the simulation. The simulation ceases (i.e., the loop is exited) after a HALT instruction has been encountered and all components have returned to their FREE states. The results that are recorded consist of the times each component spends in each state and the maximum number of entries in each queue. These times and numbers of entries are also updated each time around the main loop. Whenever the system time becomes one of those specified as a time to record the results, the results are output to the results file. The results can be used to determine which components need to be faster or replicated and which queues need to be longer.

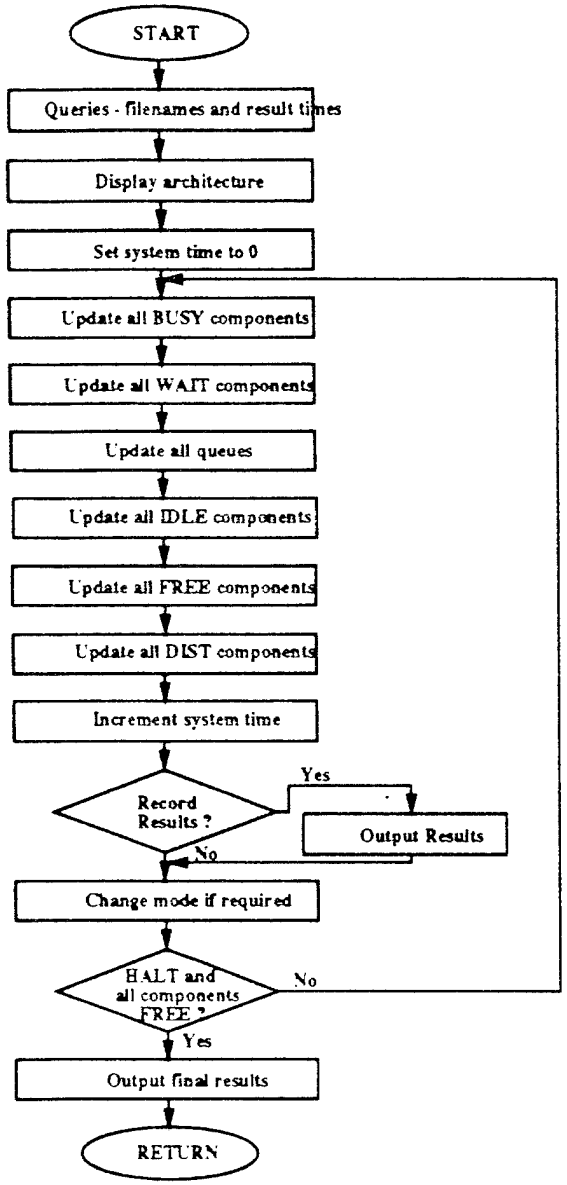


Fig. 4. Flowchart of SIMULATE program.

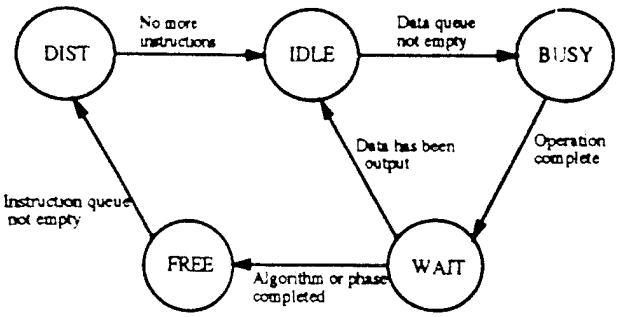


Fig. 5. Structure of state diagrams.

Before entering the main loop, the architecture is displayed on the screen just as it is displayed by EDITOR, except that the rectangles are filled in with light blue. Each state has been assigned a color as follows: FREE is light blue, DIST is green, IDLE is dark blue, BUSY is red and WAIT is magenta. Because all components begin in their FREE states, the initial color of all components is light blue. Each time a component changes its state, its display may be updated; however, the updating of the display depends on the current mode of *SIMULATE*.

SIMULATE may be dynamically put into different modes by pressing different keys as follows: "s" for single step, "a" for automatic stepping, "d" for free-running with display updating, and "n" for free-running with no display updating. In single step mode the simulation pauses and waits for a key to be pressed each time around the loop. Automatic stepping causes a brief pause each time around the loop, but a key need not be pressed. A free-running mode includes no pauses. Free-running with display is used to identify the utilization and bottlenecks by viewing the colors as the simulation progresses. The more red the better, because red indicates a components logic is being utilized. Dark blue shows that a component is waiting for input and magenta shows it is waiting for output. SIMULATE runs at its maximum speed when free-running with no display is used. Pressing "q" causes SIMULATE to terminate.

3.4 Other Main Menu Selections

The program *DISPLAY*, which is initiated by choosing "D" from the main menu, is used to superimpose percentage results on an architecture display. The user must indicate the result file to be used and the state of interest. The percent of the time spent in the specified state is displayed on the architecture. A display of the results at the end of a simulation or displays at the intermediate times for which the results were recorded may be requested.

By selecting "G" from the main menu the user may create a fractional result file in which the times spent in the states are replaced by the percentages of times spent in the states. This selection also allows certain summary information to be appended to a summary file. The selection "A" simply causes an architecture

to be displayed. The user enters the name of the architecture file in response to a query. Typing "X" causes SIMARC to terminate.

4 MATCHING AN ALGORITHM TO AN MCAP ARCHITECTURE

Matching an architecture to a set of algorithms involves a study relating the flows, storage, and processing of the data required by the algorithms. Clearly, there is no point in increasing the speed of a processing subsystem if the current interconnections and memory hierarchy are inadequate to support the processing or vice versa. But a good balance for one algorithm may not be a good balance for a different algorithm. What is needed is a satisfactory tradeoff for the work mix expected of a system and a means of evaluating the design chosen.

Space allows only a single example, so let us consider the computation that most frequently occurs in computationally intense algorithms, matrix multiplication. Let us examine how the MCAP in Fig. 2 could be analyzed for to the algorithm $AB = C$ using the middle product method [9] where A, B and C are $n \times n$ matrices. Fig. 6 shows the data flow through the MCAP.

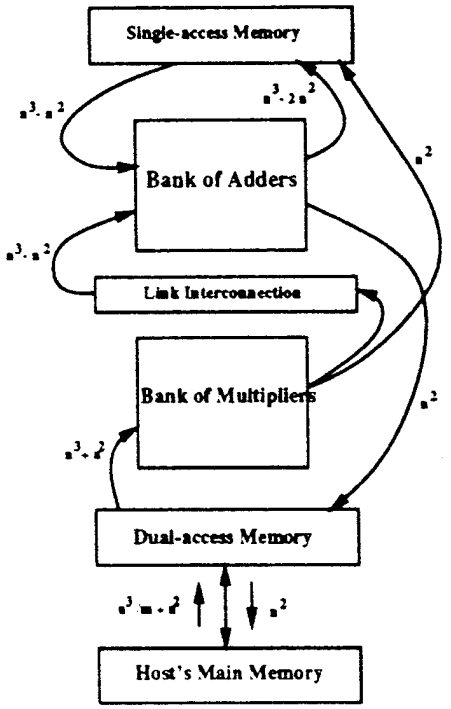


Fig. 6. Data flow for matrix multiplication using the middle product method.

The algorithm consists of the computations

$$\sum_{j=1}^{n} a_{ij} B_j = C_i \qquad i = 1, \ldots, n$$

where the a_{ij}s are the elements of A, the B_js are the rows of B, and the C_is are the rows of C. The algorithm proceeds by storing the first row of A in the D component's memory. Then the products $a_{1j}B_j$, for $j = 1, \ldots, n$, are summed and stored in the S component at the top of Fig. 2. Next, the second row of A is brought into the D component and the products $a_{2j}B_j$ are summed and the results are sent to the S component. This continues for all rows of A.

By matching this algorithm with the architecture in Fig. 2, it is seen that each adder and multiplier must perform approximately $n^3/4$ operations and each link on the left and two of the links on the right must perform approximately n^3 transfers. (The third link on the right is not needed.) The approximate number of accesses to the MCAP's S component is about $2n^3$ and the number of accesses to the D component and host's S component is about n^3. If T is the per stage processing time of the multipliers, then T should also be the per stage processing time of the adders and $T/4$ should be the transfer time of the links. The access time of the MCAP's S component should be $T/8$ for both reads and writes. For $T = 40$ ns, the link transfer time should be 10ns, the average memory access time for the MCAP's S component should be 5 ns, and the average memory access time for the D component and host's S component should be 10 ns. The computation rate would be 200 Mflops per second. If the MCAP were put into an MCM or wafer, memory interleaving were used, and multiple connections are made to the host's memory, these times would be within the capability of current HCMOS technology.

To verify the above simple analysis, we have designed a simulation program for the matrix multiplication to be run by the MCAP simulator. The instruction set for an MCAP architecture consists of two sets of instructions, internal and external. The former is processed within the instruction component and the latter is distributed to the other components. In this paper we will only discuss the external instruction set. The external instruction set consists of three types of instructions: (1) instructions which set the number of operands to be output from or input to a component, (2) instructions which set the mode of a component, and (3) instructions which set input or output connection patterns for the router components or partition and operand patterns for the controller components. When programming an MCAP architecture, each component must be programmed individually. The operation mode, input/output connection patterns, number of operands to be input or output, broadcasting connection patterns, etc., are programmed for each component using external instructions.

The operation of the leftmost link component, LINK, consists of broadcasting one element of matrix A and distributing a row of matrix B among four of the join components that are connected to the multipliers. Thus, the number of operands to be output by LINK is $N = n^2(n+1)$. The following instructions program the LINK component for matrix multiplication:

```
lmodLINK, 2                    ;set mode
lsip LINK, F077                ;set input
```

```
                                                pattern
lsbp LINK, J059, J057, J060, J062               ;set broadcast
                                                pattern
lsop LINK, #n, &, J059, J057, J060, J062        ;set output
                                                pattern
lnoo LINK, N                                    ;set number of
                                                operands out
```

The first instruction sets the mode so that the first output is to the first connection indicated by the output pattern and the next n outputs are to the connections indicated by cycling through the remainder of the output pattern. The second instruction sets the input pattern and causes all input to come from the fork, F077. The third instruction sets the broadcasting pattern by specifiing the components to receive the broadcast operand. The fourth instruction sets the output pattern, which causes the LINK component to perform a broadcast, indicated by the symbol &, and then to distribute n operands to the components listed in the instruction. This pattern is repeated until all operands have been output.

The processing components, T and E, can be programmed as follows. Since each multiplier and adder is composed of four pipeline stages, one T component followed by three E components are needed to configure one multiplier or adder. The following instructions assume $M = n^3/4$, $K = n^2$ and $L = n/4$ and set up the T component for one of the multipliers:

```
tmod T013, 48    ;set mode
trep  T013, K    ;set number of repetitions
tnoo  T013, L    ;set number of output operands
```

The mode of the T component is set so that the first operand is latched and multiplied by the next L inputs. The second instruction causes the L multiplications to be repeated K times.

5 SUMMARY AND CONCLUSION

Preliminary work indicates that MCAP component definitions when used in conjunction with the SIMARC package do permit fast prototyping of attached processors designed for high logic utilization while executing computationally intense algorithms. It is believed that SIMARC can produce designs for which the sustainable computation rate can be made to average more than 60% of the peak rate, even for fairly diverse sets of algorithms. In the matrix multiplication example presented, the average rate was 95% of the peak rate. Someone experienced with EDITOR could produce the architecture shown in Fig. 1 in less than two hours. However, creating the programs for simulating the algorithms is much more difficult and exacting. Minor errors can cause SIMULATE to cycle indefinitely and force the user to abort the simulation. Future plans call for a compiler for creating simulation programs that provides graphical assistance in determining the memory-to-memory pipelines. Such a program would move much of the responsibility for details from the user to the compiler and greatly reduce the chance for errors.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of Steve Senyszyn, Claudia Ayala, George Lammers, and Eric Adams to the SIMARC package. They helped write and modify several parts of the package.

REFERENCES

[1] S. Z. Pasha and E. H. Welbon, "Performance Directed Guidance Using Simulation," IBM RISC System/6000 Technology, Austin, TX: IBM *Advanced Workstation Division*, pp. 78-85, 1990.

[2] K. M. Nichols, "Performance Tools," *IEEE Software Trans.*, Vol. 24 No. 5, pp. 21-30, May, 1990.

[3] K. K. Bagchi, "Simulation of Multiple Processor Systems: The State of the Art," *Int'l. Jour. in Comp. Sim.*, Vol. 1, pp. 125-128, 1991.

[4] K. K. Bagchi and Ole Olsen, "Simulation of Multiple Processor Systems," *Int'l. Jour. in Comp. Sim.*, Vol. 3, pp. 107-110, 1993.

[5] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

[6] T. Hoshino, *PAX Computer: High-Speed Parallel Processing and Scientific Computing*, Addison-Wesley Publishing Company: Reading, Massachusetts, 1985.

[7] G. Gibson, V. Singh, S. Singh, Y.C. Liu, Y.C. Chang, and S. Cabrera, "MCM Implementation of Modularly Configured Attached Processors", *IEEE Int'l Computer Symp.*, Taiwan, Dec. 1994.

[8] H. V. D. Le and M. E. Perkowski, "Real Time Graphical Simulation of Systolic Arrays," *Proceedings IEEE Int'l. Symp. on Cir. and Sys.*, Vol. 1, pp. 171-174, 1989.

[9] F. Distante, V. Piuri, A. Aliquo, N. Chiapi, W. Fornaciari and P. Rostelli, "APES Implementation of a CAD Tool for Array Processor Design: Textual Definition Versus Graphic Description," *Microprocessing and Microprogramming*, Vol. 28, No. 1, pp. 63- 67, March, 1990.

ATTACHMENT E

DESIGN ISSUES IN A CMOS IMPLEMENTATION OF A MODULARLY CONFIGURED ATTACHED PROCESSOR[1]

J. Sanjay Singh, Buck W. Gremel,
Vijay P. Singh, and Glenn A. Gibson
Electrical and Computer Engineering Department
The University of Texas at El Paso
El Paso, Texas 79968-0523

ABSTRACT

Implementation of a novel modularly configured attached processor (MCAP) architecture was evaluated using 1 μm CMOS logic on an MCM-D. The transistor count was approximately nine million transistors, distributed on twenty-five chip dies. Delay, area, and power calculations were performed using the SUSPENS model. Rent's rule was found to be not applicable. Speed was calculated to be in the 150 MFLOPS range. The module foot print was calculated as 90 cm^2. Power dissipation per unit area was low enough to allow air cooling.

Index Terms: CMOS; Chip Design; Delay; Power Density; Multichip Module (MCM); Attached Processors.

[1]The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

1 Introduction

An attached processor is a processing system that is connected to a host computer for the purpose of very quickly executing most of the overall system's computational tasks. In such an organization,"the host is a program manager which handles all I/O, code compiling, and operating system functions, while the attached processor concentrates on arithmetic computation with data supplied by the host" [1].

Typical early attached processors were the AP-120B and FPS-164 made by Floating Point Systems, Inc., the IBM 3838, and the MATP made by Datawest, Inc. [1], [2], [3]. These attached processors all have their own data memories and transfer data between these memories and the main memories of their hosts using DMA data channels. They also include their own code memories where subprograms may be permanently stored or downloaded from their hosts. These subprograms are initiated by commands from the host and supervise the data flows from the attached processor's data memories, through the attached processor's processing elements, and back into the data memories.

In addition to quick execution, it is also desirable to execute as broad a set of algorithms as possible in order to create a more generally applicable processor. Thus, the underlying goal of the designer is to efficiently utilize the hardware for as broad a set of algorithms as possible. However, for most current designs, the average sustainable execution rates have been found to be only 5% to 20% of their peak rates, which are determined by summing the maximum computational rates of the processing elements. For example, the sustainable rates for the Cray X-MP with four processors may be as low as 5% for some algorithms [4]. Although some of the lost efficiency is necessitated by the algorithms, much of it is due to memory accessing and contention for shared resources in general, including internal buses.

In this paper we describe a modularly configured attached processor (MCAP) architecture which can attain quickness and high utilization through: (1) closely matching their architectures to the set of algorithms they are to execute, (2) overlapping of processing and memory accessing by using memory prefetching, (3) minimizing the movement of data, (4) using a high-speed CMOS with one micron technology, and (5) having the whole MCAP on a single MCM-D.

In Section 2, the MCAP architecture [5], [6] is summarized. Next, in Section 3, we evaluate various design options for implementing this MCAP architecture in CMOS logic.

2 MCAP Organization

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components [5]. This standard set consists of three types of asynchronous connections and twelve type of components. These component types are such that each member of the class may include parallel processing, memory to memory pipelines, and be constructed in a building block fashion. They encompass routing as well as memory, control and processing components. By overlapping processing with memory accessing and matching an architecture with a set of algorithms, it is predicted that the average sustainable rate for a specific set of algorithms can attain at least 60% of the peak rate. The definitions of the connections and component types provide a standard set of rules. These rules allow the components to be easily configured in different ways, thereby allowing the construction of attached processors that efficiently perform different sets of algorithms.

Much of the MCAP's efficiency is gained by distributing the next instructions to the various components while the current instruction is being executed. Once the algorithm begins, these instructions dictate the modes, routing patterns, prefetching patterns, etc. of the components receiving them. Once an algorithm starts, each component operates more or less on its own, except for responding to its handshaking signals. Efficiency is further enhanced by prefetching operands from the memory subsystem.

An MCAP operates by drawing an instruction stream from the memory component into the instruction component. The instruction component uses internal instructions in the stream to form external instructions that are then distributed to the other non-memory components through the MCAP's bus component. All components in the instruction stream include input instruction queues. When the non-memory components have received all of the instructions needed to perform an algorithm, they automatically prefetch the data from the memory components, route the data to and from the processor components and store the results back into the

memory components. All non-memory components have input data queues. Some controller components, which are the components that supervise all memory accessing, are used to automatically transfer data between the host's main memory and the MCAP's memory components. The instruction and data streams are separate, thereby allowing the instructions needed for the next algorithm to be distributed while the current algorithm is executing.

The three types of connections are referred to as memory, instruction, and data connections. Instruction transfers are made through instruction connections. Memory components are connected to their controller components using memory connections and all other data transfers are made by means of data connections. All connections are asynchronous and, therefore, must include handshaking lines as well as data and, perhaps, address lines.

The twelve types of components are divided into six categories: (1) Instruction, (2) Bus, (3) Memory, (4) Processor, (5) Router, and (6) Controller. The processor components can be subdivided into elementary processors (one input, one output), two-input processors, and comparators. The router components are subdivided into joins (multiple inputs, one output), forks (one input, multiple outputs), and links (multiple inputs, multiple outputs). The controller components consist of RAM controllers, single access controllers, and dual access controllers. The dual access controllers connect to main memory.

An example architecture is given in Figure 1. Its processing subsection includes a comparator (C-component), a negator (elementary component), a reciprocator (elementary component), a set of four pipelined adders capable of accumulation (via feedback), and a set of four pipelined multipliers. Each adder and multiplier is constructed of four stages (a two-input component followed by three elementary components). All communications to and from the processing components are through six link components, three on each side of the processor. Join and fork components are provided to allow flexible use of the link components. There is a dual access component to provide intermediate memory and a connection to main memory. The single access component provides internal storage.

In order to efficiently use the available logic and interconnections, an architecture must be carefully matched to an algorithm or a set of algorithms. This involves a study relating the flows,

storage and processing of the data required by the algorithm(s). Clearly, there is no point in increasing the speed of a processing subsystem if the current interconnection delays and memory hierarchy are inadequate to support the processing (or vice versa). But a good balance for one algorithm may not be equally good for a different algorithm. What is needed is a satisfactory tradeoff for the work mix expected of a system and a means of evaluating the design parameters chosen.

Next we present the design considerations in implementing the architecture of Figure 1 in CMOS logic.

3 Design Considerations and Results

In our evaluation, CMOS was picked as the benchmark logic technology because of its commercial maturity. In the future, we plan to evaluate other faster technologies such as GaAs, BiCMOS, and ECL. As for the interconnection and packaging, Deposited Interconnect Multichip Module technology (MCM-D) was picked as the preferable vehicle for implementation. Since the signal delays associated with the PCB implementation are expected to be prohibitively excessive, it was decided that the fabrication of an MCAP with MCM or Wafer Scale Integration are the only realistic alternatives for attaining high performance. Further, multichip modules are classified according to the substrate technology: MCM-Ceramic, MCM-Deposition, and MCM-Laminated. In the physical design, we must face the traditional problems in placement and routing required by the high performance systems. As the clock frequency is increased, we need to account for transmission line effects due to long interconnections. Parasitics on the interconnects, inductances on the power lines, and the I/O pin limitation are the three vital shortcomings of current packaging technologies, which could be tackled by (a) minimum chip to chip interconnections, (b) high interconnection density, and (c) parallel architecture. Other factors which need to be considered are ground and power plane generation and physical design verification. The thermal considerations are a direct result of the substrate type, bonding selection, and the placement of chip dice.

A system level model, referred to as the SUSPENS model (Stanford University System

PErformaNce Simulator) [7] is used to predict the performance of the MCAP. This model emphasizes the interactions among devices, circuits, logic, packaging, and architecture. The same model could be used to compare logic technologies (e.g. CMOS, Bipolar, and GaAs) and various packaging technologies (e.g. MCM-D, MCM-C, PCB, and WSI).

3.1 Chip Level Design

3.1.1 Transistor Count

To illustrate the method used to estimate the total number of transistors in the example MCAP (Fig. 1), we will consider one of the floating point adders. Each adder has four pipelined stages and uses the IEEE double precision standard. Further this adder could be broken down into: (a) nine 64-bit registers with 4032 transistors, (b) seventy-four 2-input XOR gates with 592 transistors, (c) one hundred and twenty-six 2-to-1 MUXs with 504 transistors, (d) two 11-bit adders with 528 transistors, (e) one 52-bit adder with 1248 transistors, (f) a 64-bit leading zero detector with 5000 transistors, (g) two 52-bit barrel shifters with 4000 transistors, and (h) rounding and other control logic taking 6500 transistors.

Thus the total number of transistors for the above adder is approximately 23K. Similarly, the transistor count for the pipelined 64-bit floating point multiplier is approximately 58K (based on a modified Booth's algorithm). Likewise, the transistor count for the other elements in the MCAP were calculated and the results are presented in table 1. The resulting number of transistors for the whole MCAP is approximately nine million.

3.1.2 Output Driver Design

In the proposed MCAP architecture, the bottle neck is the communication through link, single-access, and dual-access components because of their high fanout and large interconnection lengths. This means that the output buffers for these elements must be relatively large. We present the delay, area, and power dissipation calculations for the buffers as functions of fanout (F) and interconnection length (ℓ).

For the chip level model, we have assumed the following:

1. The input capacitance of a gate (including the lead and ESD capacitances) is C_{in} = 1 pF.
2. The width of the metal conductor used for an interconnection is W_{int} = 2.0 μm.
3. The capacitance of the interconnections is C_{int} = 2.0 pF/cm.
4. The resistance of the interconnections is R_{int} = 375 Ω/cm.
5. The feature size is L_{eff} = 1 μm.

Additional parameters can be found in Table 2.

Average Delay

In general, the minimum size of a logic gate has a W/L ratio of 2. Therefore, we began with a ratio of 2 and, by stages, moved to higher values in order to drive a load in a small amount of time. By dividing the buffering stages into the number of buffers with increasing W/L, optimum speeds can be achieved. It has been found that a stage ratio of e [8] gives best results. We have used a stage ratio of 3 for simplicity. The optimum number of stages (N) is dependent on the load capacitance (C_l). The relationship is

$$N = 0.91(\ln C_l + 4.19)$$

where N is truncated (rounded down) to the nearest integer.

Using the optimum number of stages, the average delay is

$$T_{avg} = 0.484(N - 1) + 5C_l/3(N - 1) + 0.076 \text{ ns}$$

Delay calculations are shown in Fig. 2.

Buffer Area

A simple inverter with $(W/L)_n = (W/L)_p = 2$ will need an area of 66 μm^2. A buffer with equal rise (t_r) and fall (t_f) times requires $(W/L)_p = 2(W/L)_n = 4$ and the area is going to be

101 μm^2. The total area of the buffer depends on the number of stages and, hence, is a function of F and ℓ. We have

$$Area = 66 + 3[14(N-1) + 36(1 + 3 + 3^2 + \cdots + 3^{N-2})] \approx 55 \times 3^{N-1} \ \mu m^2$$

Area calculations are plotted in Fig. 3.

Power Dissipation In The Buffer

In CMOS, most of the power is dissipated during switching and, hence, dynamic power is approximately equal to the total power. The dynamic power is

$$P_d = C_T \times v^2 \times f_{avg} = 3.3^2(C_l + C_{buff})/T_{avg}$$

where $C_{buff} = 0.0152(3^{N-1})$ pF.

Power dissipation calculations are presented in Fig. 4.

Since the design of an MCAP uses asynchronous communication, the transfers over a link component involve the return of an acknowledge signal and the transmission of an output enable signal. It is estimated that the transfer rate may be as high as $f = 1/2[T_{avg} + T_{chip}]$ Hz (where T_{avg} is the delay on the interconnection and T_{chip} is the chip delay).

Load Capacitance

For the load capacitance

$$C_l = C_{int_{tot}} + F \times C_{in},$$

with

$$C_{int_{tot}} = C_{int} \times \ell = \ell \text{ pF}$$

where ℓ is in cm and $C_{int} = 1$ pF/cm. Therefore,

$$C_l = (\ell + F) \text{ pF}.$$

The resistance of the interconnect is

$$R_{int_{tot}} = R_{int} \times \ell = 3.4 \times \ell \ \Omega .$$

3.1.3 Modified SUSPENS Model

Given (a) the approximate number of logic gates, (b) the transistor technology parameters, (c) packaging technology parameters, and (d) the number of pads per chip (estimated from Rent's rule), the SUSPENS model [7] can estimate system performance. Rent's rule is an empirical result obtained by observing existing designs. The design philosophy and methodology affect Rent's constants. If the predictions are made for a system with an entirely different design philosophy from the one from which Rent's data were obtained, the results will have little meaning. The SUSPENS model, as originally proposed, used Rent's constants, therefore we have developed constants that are applicable to the novel architecture of the MCAP.

Our approach to developing these constants was to determine the area needed for an inverter (with equal rise and fall times) and a carry generator circuit. From this we computed the average area per transistor. Thirty percent of that area was assumed to be taken by the interconnections, resulting in a figure of 50.7 μm^2 per transistor. This result was used as input to the SUSPENS model for computations done at the chip level. The I/O buffer areas were estimated separately as presented in section 3.1.2.

To illustrate the use of the SUSPENS model (using table 2), we will consider the adder chip (see Fig. 5). The adder chip contains four 64-bit floating point adders. The input stage delay is

$$T_i = f_g \cdot \frac{R_{tr}}{K_i} \cdot 3K_oC_{tr}$$

f_g (number of n − transistors in series) = 3

K_i (W/L ratio of input transistors) = 4

K_o (W/L ratio of output buffer) = 4

C_{tr} (for 1 μm CMOS) = 3 fF

R_{tr} (for 1 μm CMOS) = 15 K

$$T_i = 0.41 \text{ ns}$$

The output stage delay is

$$T_o = \frac{f_gR_{tr}}{K_o} \cdot [\ell_{av}C_{int} + K_iC_{tr}] + \ell_{av}R_{int}[\frac{\ell_{av}C_{int}}{2} + K_iC_{tr}]$$

ℓ_{av} (average interconnection length)

$C_{int} = 2$ pF/cm

$R_{int} = 375\ \Omega$/cm

The total gate delay, $T_g = T_i + T_o$, is 0.84 ns. The delay for an adder is

$$T_{chip} = f_{ld}T_g + R_{int}C_{int}(D_c^2/2) + (D_c/v_c)$$

restrict the logic depth, f_{ld}, to 6

$v_c = 2.5 \times 10^{12}$ cm/sec

$D_c = 0.3$ cm

$$T_{chip} = 6(0.84 \times 10^{-9}) + 375(2 \times 10^{-12})(0.3^2/2) + (0.3/2.5 \times 10^{12})$$

$= 5.07$ ns (which includes the latch time, logic time, setup time, and clock skew)

Thus the maximum frequency of the adder chip is 197 MHz. By incorporating pipelining and recalling that the total chip area actually has four of these floating point 64-bit adders, the throughput is improved by a factor of more than four.

The power dissipation for the adder chip is found as follows. The external capacitance of a gate is

$$C_{ext} = f_g \ell_{avg} C_{int} + f_g K_i C_{tr} = 36.5 \text{ fF},$$

where $\ell_{avg} = 81.3 \times 10^{-6}$ cm. The internal capacitance of a gate is

$$C_{int} = 3K_o C_{tr} + 5C_{tr} = 51 \text{ fF}$$

The total capacitance per logic gate, $C_g = C_{ext} + C_{int}$, is 87.5 fF.

The dynamic power dissipation per gate is based on the maximum adder chip operating frequency (f_c) and the percentage of gates that switch during a clock period (f_d). The dynamic power dissipation is

$$P_g = \frac{1}{2} f_c f_d C_g V_{DD}^2$$

$$= \frac{1}{2}(131 \times 10^6)(0.3)(83.5 \times 10^{-12})(3.3)^2$$

$= 56.3\ \mu$W

The power dissipation of the chip is the product of the number of gates (N_g) and the dynamic power dissipation per gate (P_g).

$$\begin{aligned} P_c &= N_g \cdot P_g \\ &= (176k/4) \cdot (56.3 \times 10^{-6}) \\ &= 1.65\text{W} \end{aligned}$$

Thus the power density for the adder chip (area = 0.09 cm^2) is 18.4 W/cm^2.

3.2 Interconnection and Packaging Considerations

Once the results for each chip have been obtained (see Table 4), the package level model is incorporated using thin film hybrid parameters (Table 3). The average interconnection length at the module level (in units of chip footprint size) is [7]

$$\begin{aligned} \bar{R}_m &= \frac{2}{9}[7\frac{N_c^{\eta-0.5}-1}{4^{\eta-0.5}-1} - \frac{1-N_c^{\eta-0.75}}{1-4^{\eta-0.75}}]\frac{1-4^{\eta-1}}{1-N_c^{\eta-1}} \\ & N_c = 4 \\ & \eta = 0.65 \\ \bar{R}_m &= 1.33 \end{aligned}$$

To find the number of interconnections for the adder chip, we assume that the output buffers have a fanout of 3 and a critical length of 1.5 cm. Thus, from Fig. 3, we see that the buffer delay is 2.45 ns. Further, the capacitive delay (caused by the loading of the I/O pins and contact pads) and the time of flight delay on the transmission lines are both calculated using

$$T_{additional} = 2 \cdot Z_o \cdot C_{pad} + L_{int}/v_m = 0.124 \text{ ns}$$

Adding in T_{chip} = 5.07 ns, we determine that the total delay of the adder chip = 5.07 ns + 2.45 ns + 0.124 ns = 7.64 ns. Thus, the adder chip can output at the rate of 131 MHz.

The area required by the output buffers are added to the transistor area to get the die area. Assuming an area distributed solder bumps with 100 μm diameter and 250 μm pitch. The footprint of the Adder chip die is found to be 0.64 cm.

The results from repeating this process for the other chips are presented in Table 4 (Note that all of the chip areas have been optimized for a power density of no more than 15 W/cm^2). Hence, we determine that the module frequency is approximately 150 MFLOPS (considering the processing and driving involved in one cycle). The module size is 9.0 cm x 10.0 cm. Module power dissipation (P_m) is determined by

$$\begin{aligned} C_m &= \frac{F_c}{1+F_c} N_c N_p (3\frac{1-5^N}{1-5} C_{tr} + 2C_{pad} + \bar{R}_m F_p C_{int}) \\ &= \frac{4}{5} 25(600)[3\frac{1-5^5}{1-5} 3 \times 10^{-15} + 2(0.25 \times 10^{-12}) + 1.33(0.64)(1 \times 10^{-12})] \\ &= 0.1\ \mu\text{F} \\ P_m &= \frac{1}{2}(F_D)(f_s)(C_m)(V_{DD}^2) \\ &= \frac{1}{2}(0.5)(100 \times 10^6)(0.1 \times 10^{-6})(3.3^2) \\ &= 27.4\ \text{W} \end{aligned}$$

The actual power dissipation is the greater of P_m and the sum of the power dissipated at all the chip dice. Thus we report the power dissipation for the module to be 68.82 W.

4 Conclusions

Design evaluation for implementing a novel modularly configured attached processor architecture using 1 μm CMOS logic on an MCM-D (see Fig. 6) revealed that approximately nine million transistors will be needed. These could be placed on a set of twenty five chip dies. Delay, area, and power calculations were done with the SUSPENS model (however, Rent's rule was not used). Delay calculation (including logic delay, interconnect delay and output driver delay) showed that the MCAP module, on average, would achieve speeds in the 150 MFLOPS range. The single-access memory controller component chip (S-control) was found to be the slowest (110 MHz). Higher speeds would be achievable with faster logic like BiCMOS, ECL, and GaAs. In fact, further calculations using 0.5 μm CMOS and 0.25 μm CMOS logic show that average speeds of the S-control chip increase from 110 MHz (for 1 μm design rule) to 160 MHz (for 0.5 μm design rule) and 220 MHz (for 0.25 μm design rule).

Power dissipation calculation showed that approximately 70 watts will be dissipated in the MCAP module and air cooling would suffice for the 1 μm CMOS design rule.

We are in the process of performing further design calculations involving wafer-scale integration (WSI) and GaAs technology.

References

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1985.

[2] J. A. Swanson, G. R. Cameron, and J. C. Haberland, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *IEEE Computer*, vol. 16, no. 6, pp. 85–91, June 1983.

[3] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

[4] J. H. Tang and E. S. Davidson, "An Evaluation of Cray I and Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," *Proc. 1984 Int'l Conf. on Supercomputing*, pp. 510–518, 1988.

[5] G. A. Gibson, "Investigation of Modularly Configured Attached Processors with Intelligent Memories," *Technical Report to Office of Naval Research*, (Grant No. N00014-93-1-1343), March 31, 1994.

[6] G. A. Gibson, "Application and implementation of a modularly configurable attached processor," *International Symposium on High-Performance Computer Architecture*, 1994.

[7] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, 1990.

[8] K. E. N. Weste, *Principles of CMOS VLSI design - A Systems perspective*, Addison Wesley, 1993.

Captions to illustrations:

1. Fig. 1. An example MCAP architecture.
2. Fig. 2. Buffer and interconnect delay.
3. Fig. 3. Buffer area.
4. Fig. 4. Power dissipated in the Buffer.
5. Fig. 5. Block diagram of the Adder chip.
6. Fig. 6. Layout of the MCAP on an MCM.

ELEMENT	DESCRIPTION	# TRANSISTORS
Memory Element	has 4K each of RAM and ROM	1.84 M
Instruction	has 8 words of FIFO	10.0 K
Bus	has 8 words of FIFO	10.0 K
Elementary	has 8 words of FIFO	10.0 K
Two input	has 8 words of FIFO	12.0 K
Join	3 inputs and 8 FIFO	14.0 K
Fork	3 outputs and 8 FIFO	07.0 K
Link	4 inputs and 5 outputs	15.0 K
Static Ram	16 elements of 1 K each	06.3 M
Single Access Controller	Controls 8 memory elements	11.0 K
Dual Access Controller	Controls 8 memory elements and 6 DMA channels	71.0 K
Compare	sends out Flags and Indices	25.0 K
Reciprocate	using Convergence method	50.0 K
Negate	invert the sign bit	01.0 K
Floating Point Adder	using CLAs, Barrel shifters etc.,	23.0 K
Floating Point Multiplier	using modified Booth's algorithm	61.0 K
MCAP	has 12 types of components	9 Million
MCM	with 25 chips and 600 I/Os	9 Million

Table. 1 Transistor count for the various MCAP components

Parameter	CMOS
L_{eff} (μ)	1.0
t_{gox} (Å)	250
V_{dd} (v)	3.3
R_{tr} (ohm)	15 K
C_{tr} (fF)	3.0
W_{int} (μ)	2.0
W_{sp} (μ)	2.0
H_{int} (μ)	0.4
p_w (μ)	4.0
n_w	3
R_{int} (ohm/cm)	375
C_{int} (pF/cm)	2.0

Table 2. 1μ Technology parameters

Parameter	MCM-D
P_w (μ)	50
N_w	2
W_{int} (μ)	25
W_{sp} (μ)	25
H_{int} (μ)	2.0
R_{int} (ohm/cm)	3.4
Diel. const.	3.4
V_m (cm/nS)	16
C_{int} (pF/cm)	1.0
Z_0 (ohm)	60
C_{pad} (pF)	0.25
P_p (μ)	100

Table 3. Thin film hybrid parameters

Parameters → / Component ↓	# Transistors	# of I/O'	Latency (nS)	Power Diss. (W)	Area (cm^2)
Instn/Bus	21.2 k	200	7.40	1.65	0.14
Link	16.2 k	200	8.16	1.27	0.14
S-control	32.7 k	222	8.76	0.70	0.14
D-control	85.4 k	846	8.34	3.67	0.56
ROM-4k	262.6 k	80	9.07	1.66	0.11
SRAM-4k	1.57 M	150	9.07	8.66	0.58
SRAM-1k	393.6 k	150	9.07	2.36	0.16
Multiplier	330 k	660	8.17	3.44	0.41
CNR	120 k	874	7.19	2.54	0.55
Adder	178 k	660	7.61	2.75	0.41
MODULE	9 Million	600	—	68.8	90

Table 4. Various parameters of the MCAP on an MCM-D using 1μ CMOS process

(optimized for Q ≤ 15 W/cm²)

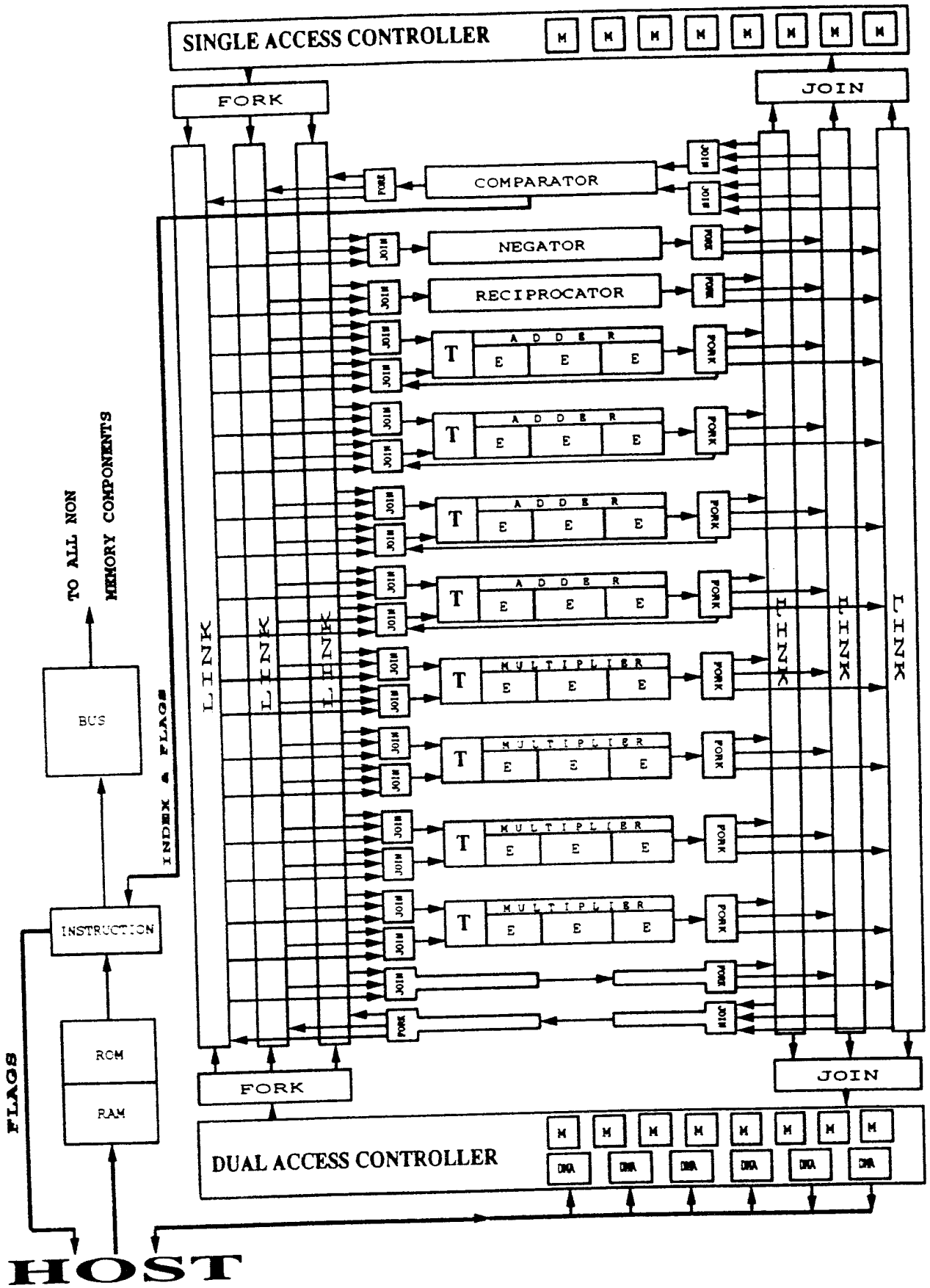

SINGLE ACCESS CONTROLLER
FORK
JOIN
COMPARATOR
NEGATOR
RECIPROCATOR
ADDER
ADDER
ADDER
ADDER
MULTIPLIER
MULTIPLIER
MULTIPLIER
MULTIPLIER
T
E
LINK
LINK
LINK
LINK
LINK
TO ALL NON
MEMORY COMPONENTS
INDEX & FLAGS
BUS
INSTRUCTION
ROM
RAM
FLAGS
FORK
JOIN
DUAL ACCESS CONTROLLER
M
DMA
HOST

Fig. 1

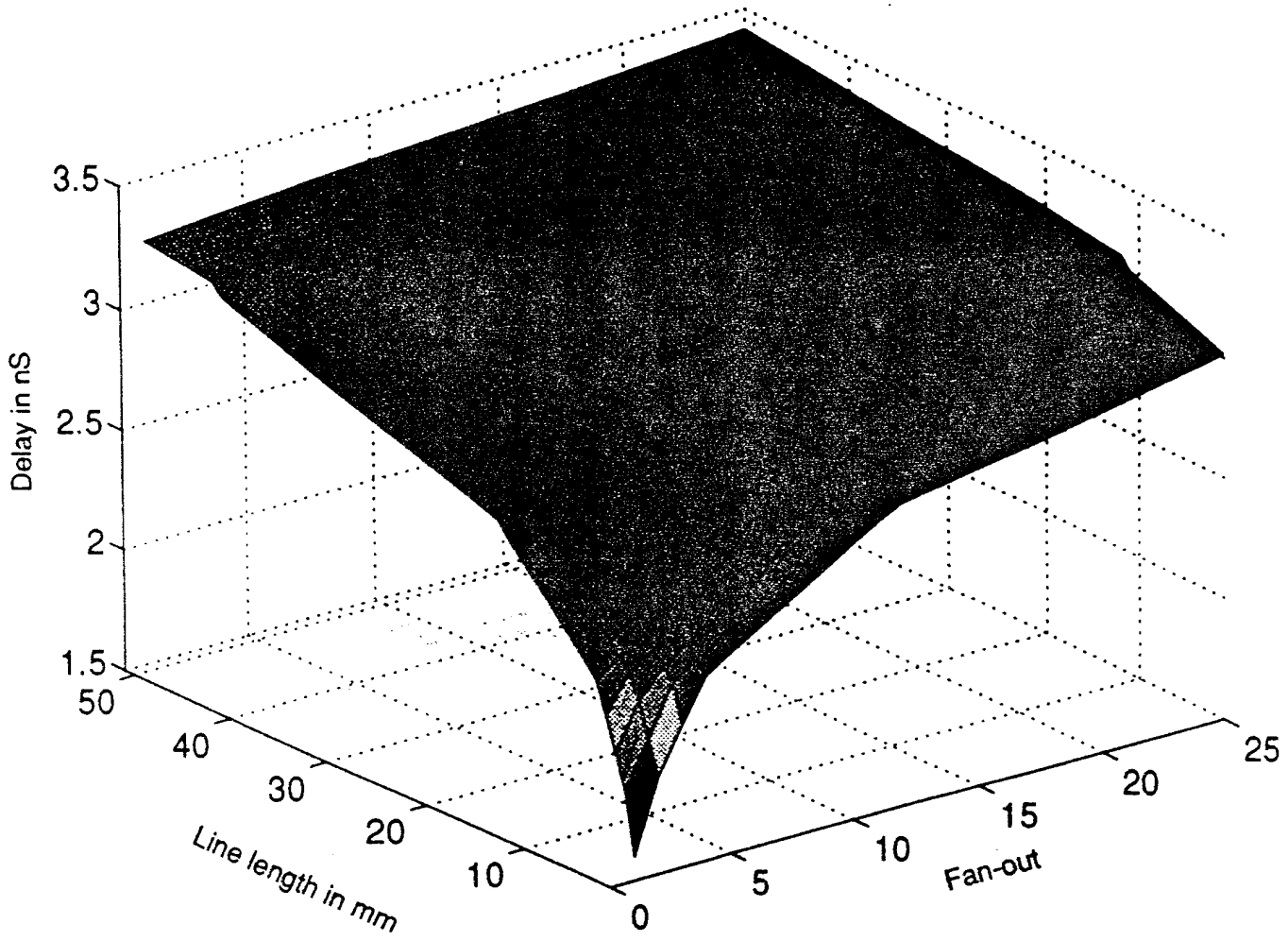

3.5
3
2.5
2
1.5
Delay in nS
50
40
30
20
10
Line length in mm
0
5
10
15
20
25
Fan-out

Fig. 2

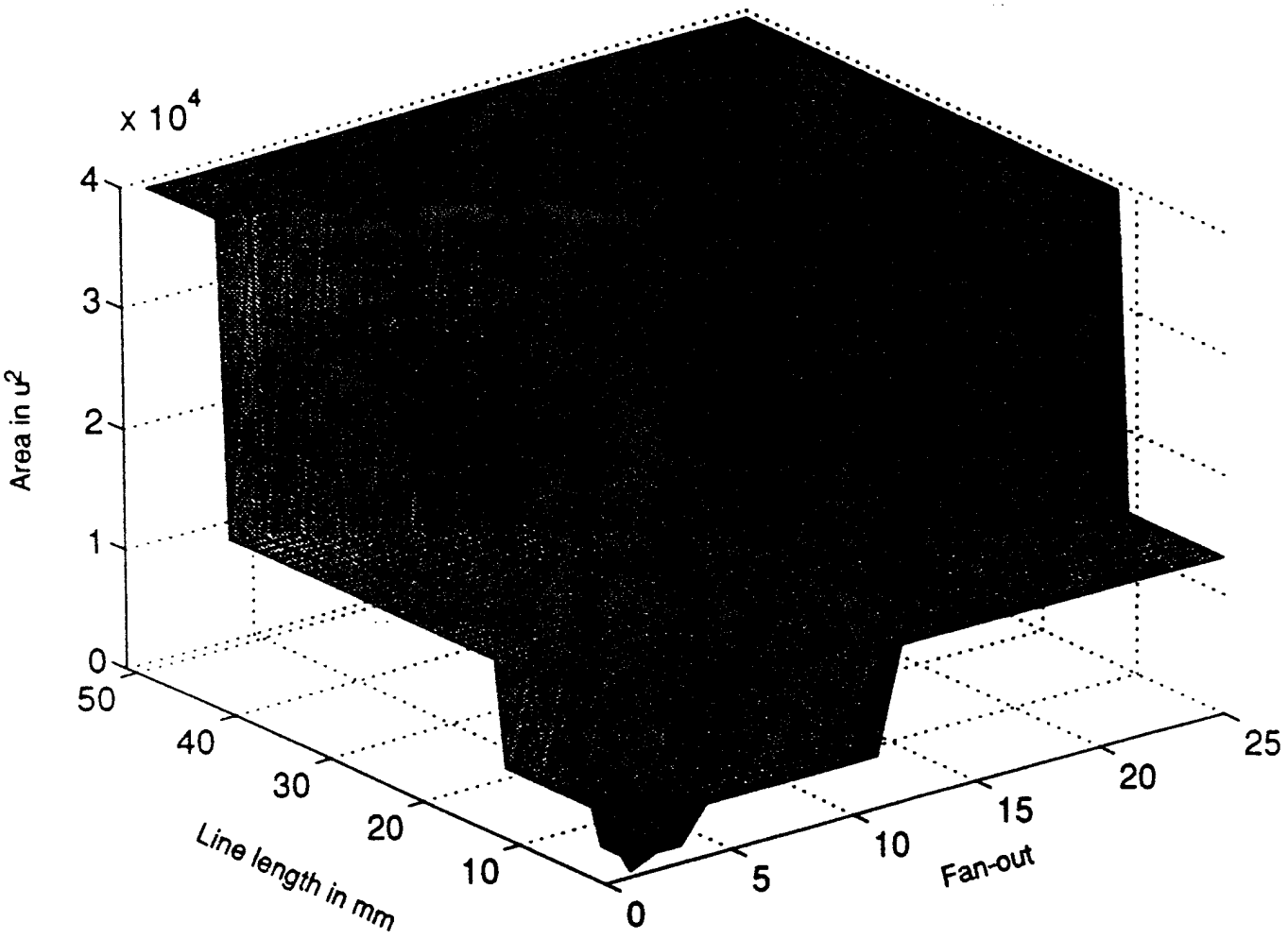

x 10^4
4
3
2
1
0
Area in u^2
50
40
30
20
10
0
Line length in mm
0
5
10
15
20
25
Fan-out

Fig. 3

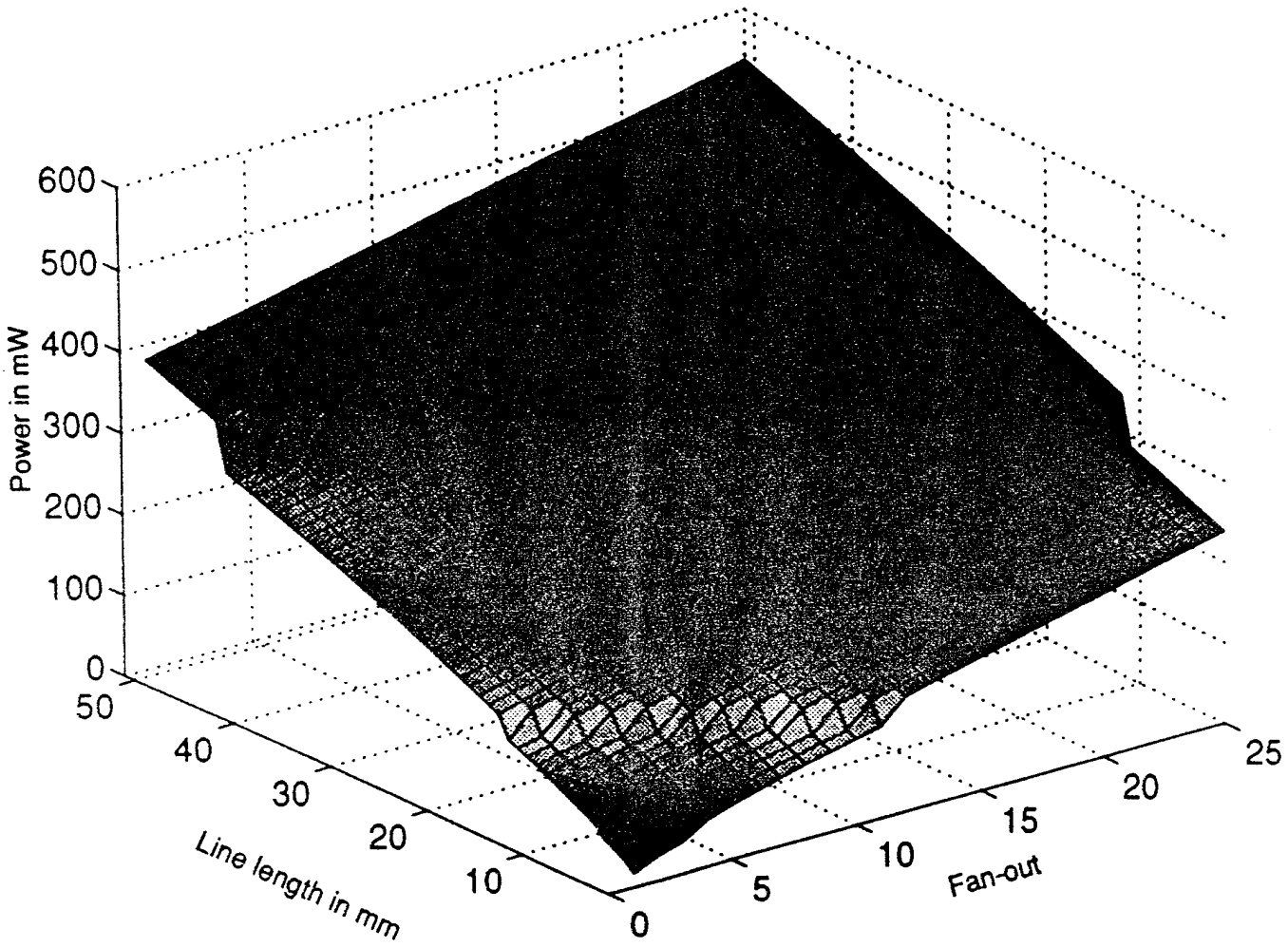

600
500
400
300
200
100
0
Power in mW
50
40
30
20
10
0
Line length in mm
0
5
10
15
20
25
Fan-out

Fig. 4

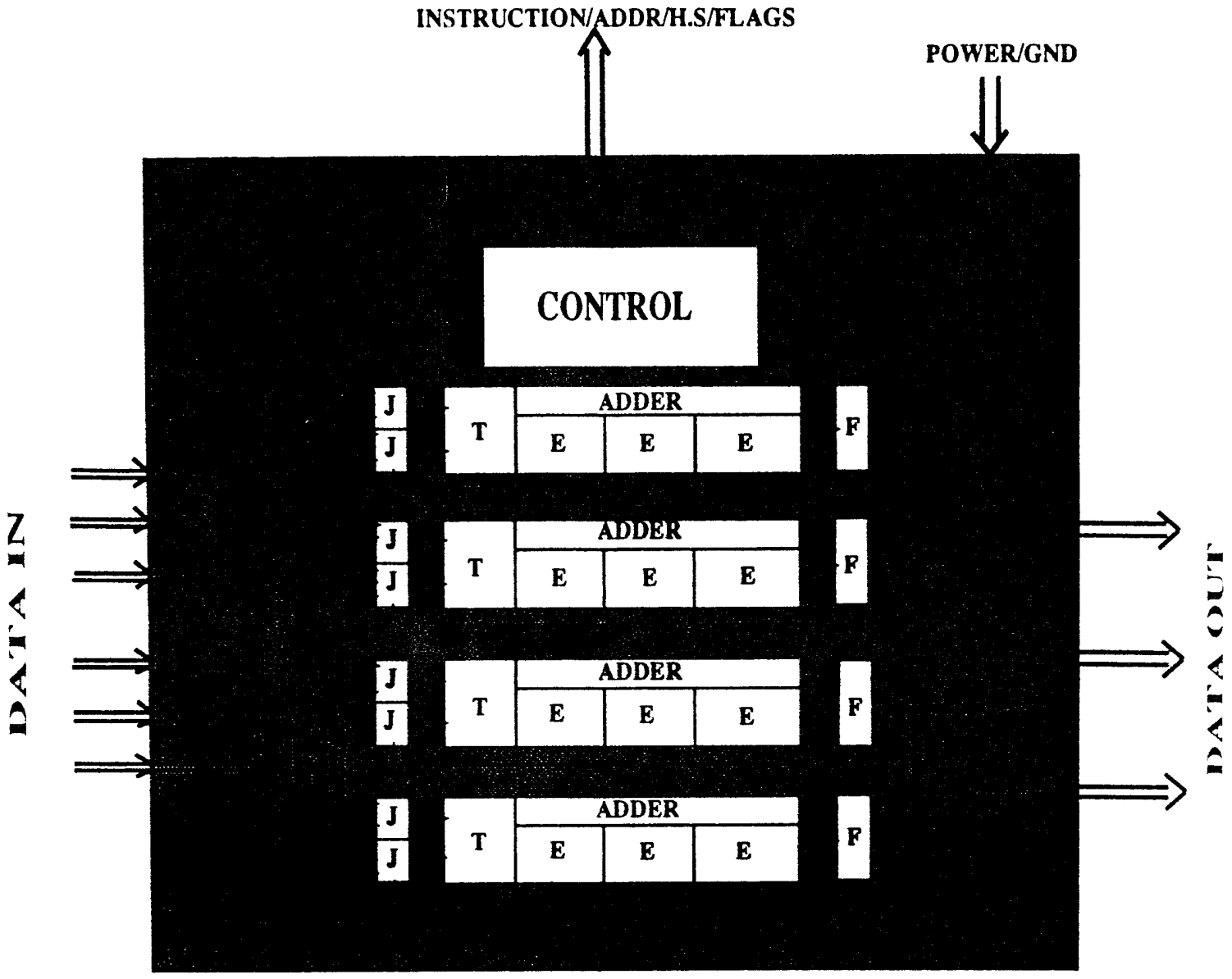

INSTRUCTION/ADDR/H.S/FLAGS
POWER/GND
CONTROL
J
J
T
ADDER
E
E
E
F
DATA IN
DATA OUT

Fig. 5

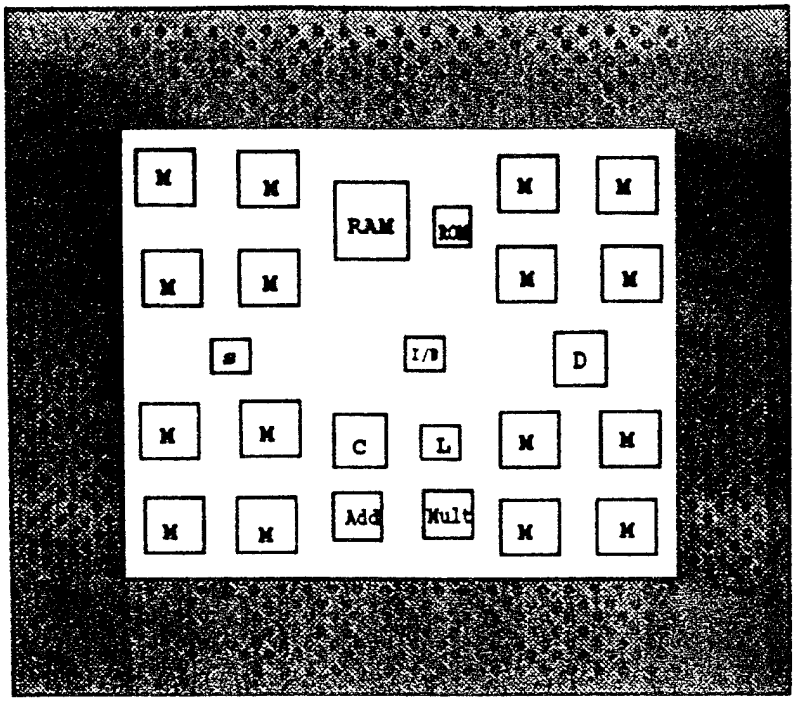


Fig. 6